Proportionate gradient updates with PercentDelta

Sami Abu-El-Haija
Google Research
Mountain View, CA
haija@google.com

Abstract

Deep Neural Networks are generally trained using iterative gradient updates. Magnitudes of gradients are affected by many factors, including choice of activation functions and initialization. More importantly, gradient magnitudes can greatly differ across layers, with some layers receiving much smaller gradients than others, causing some layers to train slower than others and therefore slowing down the overall convergence. We analytically explain this disproportionality. Then we propose to explicitly train all layers at the same speed, by scaling the gradient w.r.t. every trainable tensor to be proportional to its current value. In particular, at every batch, we want to update all trainable tensors, such that the relative change of the L1-norm of the tensors is the same, across all layers of the network, throughout training time. Experiments on MNIST show that our method appropriately scales gradients, such that the relative change in trainable tensors is approximately equal across layers. In addition, measuring the test accuracy with training time, shows that our method trains faster than other methods, giving higher test accuracy given same budget of training steps.

1 Introduction

Weights of Deep Neural Networks (DNNs) are commonly randomly initialized. The $j$-th layer’s weight matrix $W_j$ is initialized from a distribution that is generally conditioned on the shape of $W_j$.

Feed-forward DNNs generally consist of a series of matrix multiplications and element-wise activation functions. For example, input vector $x_0$ is transformed to the output vector $x_L$ by an $L$-layer neural network $h_{θ}(x_0) = x_L$, depicted as:

\[
x_0 \rightarrow W_0 \times \sigma \rightarrow W_1 \times \sigma \rightarrow \cdots \rightarrow W_{L-1} \times \sigma \rightarrow x_L
\]

Where $\times$ is a matrix-multiply operator, $\sigma$ is an element-wise activation (e.g. logistic or ReLu). The DNN parameters $θ = \{W_0, W_1, \ldots, W_{L-1}\}$ are optimized by a training algorithm, such as Stochastic Gradient Descent (SGD), which iteratively applies gradient updates:

\[
W_j^{(t+1)} := W_j^{(t)} - \eta \gamma(t) \frac{∂J}{∂W_j},
\]

where $W_j^{(t)}$ is the value of the $j$-th layer weight matrix $W_j$ at timestep $t$, $\eta \in \mathbb{R}$ is the learning rate, the decay function $\gamma : \mathbb{R} \rightarrow [0, 1]$, generally decreases with time $t$, and $\frac{∂J}{∂W_j}$ is the partial gradient of training objective $J$ w.r.t. $W_j$, evaluated at time $t$ for a batch of training examples. For notational convenience, we define the delta SGD as: $\Delta_{SGD}W = \eta \gamma(t) \frac{∂J}{∂W}$.
Under usual circumstances, the magnitudes of gradients can widely vary across layers of a DNN, causing some trainable tensors to change much slower than others, slowing down overall training. Several proposed methods mitigate this problem, including utilizing: per-parameter adaptive learning rate such as AdaGrad (Duchi et al., 2011) or Adam (Ba and Kingma, 2015); normalization operators such as BatchNorm (Ioffe and Szegedy, 2015), WeightNorm (Salimans and Kingma, 2016), or LayerNorm (Ba et al., 2016); and intelligent initialization schemes such as xavier’s (Glorot and Bengio, 2010). These methods heuristically attack the disproportionate training problem, which we justify in section 2. In this paper, we propose to directly enforce proportionate training of layers. Specifically, we propose a gradient update rule that moves every weight tensor in the direction of the gradient, but with a magnitude that changes the tensor value by a relative amount. We use the same relative amount across all layers, to train them all at the same speed.

The remainder of the paper is organized as follows. In Section 2 we illustrate the disproportionate training phenomena on a (toy) hypothetical 4-layer neural network, by expanding gradient terms using BackProp (Rumelhart et al., 1986). In Section 3 we summarize related work. Then, we introduce our algorithm in Section 4. We show experimental results on MNIST in Section 5. In Section 6, we discuss where PercentDelta could be useful and potential future direction. Finally, we conclude our findings in Section 7.

2 Disproportionate Training

We use a toy example to illustrate disproportionate training across layers. Assume a 4-layer network with trainable weight matrices \( \{ W_0, W_1, W_2, W_3 \} \) and output vector \( x_4 \). The gradient of the objective w.r.t. output vector \( x_4 \) can be directly calculated from the data e.g. using cross-entropy loss. We write-down the gradient of the objective \( J \) w.r.t. the last layer’s weight matrix \( W_3 \) as:

\[
\frac{\partial J}{\partial W_3} = \left[ \sigma'(W_3 \times x_3) \circ \frac{\partial J}{\partial x_4} \right] \times x_3^T,
\]

where \( \sigma'(\cdot) \) is the derivative of \( \sigma(\cdot) \) w.r.t. its input, and \( \circ \) is the Hadamard product.

We also write-down the gradient w.r.t. \( W_1 \) and \( W_2 \), then expand the expressions using the Back Propagation Algorithm (Rumelhart et al., 1986):

\[
\frac{\partial J}{\partial W_2} = \left[ \sigma'(W_2 \times x_2) \circ \frac{\partial J}{\partial x_3} \right] \times x_2^T
\]

\[
= \sigma'(W_2 \times x_2) \circ W_3^T \times \left[ \sigma'(W_3 \times x_3) \circ \frac{\partial J}{\partial x_4} \right] \times x_2^T
\]

\[
\frac{\partial J}{\partial W_1} = \left[ \sigma'(W_1 \times x_1) \circ \frac{\partial J}{\partial x_2} \right] \times x_1^T
\]

\[
= \sigma'(W_1 \times x_1) \circ W_2^T \times \sigma'(W_2 \times x_2) \circ W_3^T \times \left[ \sigma'(W_3 \times x_3) \circ \frac{\partial J}{\partial x_4} \right] \times x_1^T
\]

Note the following in the above equations: First, the derivatives \( \frac{\partial J}{\partial W_3}, \frac{\partial J}{\partial W_2}, \frac{\partial J}{\partial W_1} \) look similar. In fact, they are almost sub-expressions of one another, with the exception of the right-most row-vector \( x_j^T \). Second, all quantities in brackets are column-vectors. The gradient matrix \( \frac{\partial J}{\partial W_j} \) is determined by an outer product of a column-vector (in brackets) times a row-vector \( x_j^T \).
What can we conclude about the magnitudes (e.g. the L1 norm) of $\frac{\partial J}{\partial W_3}$, $\frac{\partial J}{\partial W_2}$, and $\frac{\partial J}{\partial W_1}$? Each $\frac{\partial J}{\partial W_j}$ is calculated using three types of multiplicands: $\sigma'(\cdot)$, $W_j^T$, and $x_j^T$. Therefore, the magnitudes of $\frac{\partial J}{\partial W_j}$ are affected by:

1. The value of the derivative of $\sigma'(\cdot)$, which is evaluated element-wise. If $\sigma'(\cdot)$ is generally less than 1, then we can expect the gradient to be smaller for earlier layers than later ones, since they are multiplied by $\sigma'(\cdot)$ more times. If it is generally greater than 1, then we can expect the gradient to be larger for earlier layers. For very deep networks (recurrent or otherwise), the former situation can cause the gradients to vanish, while the latter can cause the gradients to explode. ReLu mitigates this problem, as its derivative $= 1$, in locations where input is positive.

2. The magnitude of $W_j$’s. In practice, all $W_j$’s are initialized from the same distribution. If the L1-norms of rows in $W_j^T$ are less than 1, then we should expect $\left|\left| W_j^T \sigma'(z) \right|\right| < \left|\left| \sigma'(z) \right|\right|$, yielding smaller gradients for earlier layers. Otherwise, if the row L1 norms are greater than 1, then earlier layers should receive larger gradient magnitudes. Weight normalization (Salimans and Kingma, 2016) and intelligent initialization schemes (e.g. Glorot and Bengio, 2010) can mitigate this problem.

3. The norm of the row-vector $x_j$. If in the forward pass, the activations consistently grow (e.g. unbounded activation) with subsequent layers, then later layers will receive larger gradients. BatchNorm (Ioffe and Szegedy, 2015) and LayerNorm (Ba et al., 2016) mitigate this problem.

3 Related Work

3.1 Adaptive Gradient

Duchi et al. (2011) proposed AdaGrad. A training algorithm that keeps a cumulative sum-of-squared gradients (i.e. second moment of gradients):

$$S_j^{(t)} = S_j^{(t-1)} + \left( \frac{\partial J}{\partial W_j} \circ \frac{\partial J}{\partial W_j} \right), \quad (3)$$

Then divides (element-wise) the gradient by the square-root of the sum:

$$W_j^{(t+1)} := W_j^{(t)} - \eta \gamma(t) \left( \frac{\partial J}{\partial W_j} \circ \left( S_j^{(t)} \right)^{-1/2} \right), \quad (4)$$

or equivalently $\Delta_{\text{AdaGrad}} W_j = \eta \gamma(t) \left( \frac{\partial J}{\partial W_j} \circ \left( S_j^{(t)} \right)^{-1/2} \right)$, where the $(\cdot)^{-1/2}$ power operator is applied element-wise. In essense, if some layer receives large gradients, then they will be normalized to smaller gradients through the division. However, a weakness in AdaGrad is that at some point, the $S$ will grow too large, effectively making $(S)^{-1/2} \approx 0$ and therefore slowing or halting training.

Ba and Kingma (2015) propose Adam, which keeps exponential decaying sums, of gradients and of square gradients, respectively known as first and second moments. Adam offers two benefits over AdaGrad. First, its decaying sums should not grow to infinity and therefore training should not halt. Second, the exponential-decay averaging was shown to speed up training (Momentum, Sutskever et al., 2013). For details, we point readers to the Adam paper (Ba and Kingma, 2015).

3.2 LARS

You et al. (2017) propose to normalize every layer’s gradients by the ratio of L2 norms of the parameter and the gradient. Namely, they propose the gradient update rule:

$$\Delta_{\text{LARS}} W_j = \eta \gamma(t) \left( \frac{\left|\left| W_j^{(t)} \right|\right|_2}{\left|\left| \left( \frac{\partial J}{\partial \left( W_j \right)} \right) \right|\right|_2} \right) \frac{\partial J}{\partial W_j}, \quad (5)$$

which effectively normalizes the gradient $\frac{\partial J}{\partial W_j}$ to be unit-norm. This setup is very similar to ours, with two differences: First, our norm operator is L1 rather than L2. Second, our norm operator is
applied outside the division (i.e. our division is element-wise). Our proposed algorithm, PercentDelta, was used to train our work (Abu-El-Haija et al., 2017), before we were aware of the work of (You et al., 2017). In (Abu-El-Haija et al., 2017), PercentDelta gave us 1% improvement over Adam, over all datasets. Nonetheless, we only discuss MNIST experiments in this paper, and leave graph embedding experiments for follow-up work.

4 PercentDelta

For every weight matrix \( W_j \), and similarly bias vectors, we propose the gradient update rule:

\[
\Delta_{\text{PercentDelta}} W_j = \eta \gamma(t) \left( \frac{\text{size}(W_j)}{\left\| \frac{\partial J}{\partial W_j} \right\|_1} \right) \frac{\partial J}{\partial W_j}, \tag{6}
\]

where scalar \( \text{size}(W) \in \mathbb{Z}^+ \) is the number of entries in \( W \), and the scalar:

\[
\left\| \frac{\partial J}{\partial W_j} \right\|_1 \tag{7}
\]

normalizes the gradient \( \frac{\partial J}{\partial W_j} \) of the \( j \)-th layer, so that it is more proportional to its current parameter value \( W_j(t) \), and \( \| \cdot \|_1 \) is the L1-norm. We avoid division-by-zero errors by adding an epsilon to the denominator. The divide operator within the L1 norm is applied element-wise and the outer divide operator is scalar. Note that the “gradient multiplier” fraction (Equation 7) is \( \in \mathbb{R}^+ \). Therefore, we only change the gradient’s magnitude, but not its direction.

Now, we justify Equation 6. Assume that \( W_s \) is scalar. Therefore, \( \text{size}(W_s) = 1 \) and the L1-norm becomes an absolute value. Equation 6 simplifies to:

\[
W_s(t+1) := W_s(t) - \eta \gamma(t) \left| \frac{W_s(t)}{\left\| \frac{\partial J}{\partial W_s} \right\|_1} \right| \frac{\partial J}{\partial W_s} \tag{8}
\]

where \( \text{sign}(z) = 1 \) if \( z > 0 \) and \( = -1 \) if \( z < 0 \). Therefore, \( W_s \) will change with a quantity proportional to its current value. In particular, the scalar \( \eta \gamma(t) \) determines the percentage at which \( W \) changes at timestep \( t \), giving rise to the name: PercentDelta. For example, if we setup a decay schedule on \( \gamma(t) \), such that \( \eta \gamma(t) \) changes during training from 10% to 0.01%, then the network parameters will change at a rate of 10% in early mini-batches, and gradually decrease their PercentDelta updates to 0.01% towards end of training, consistently across all layers, regardless of the current parameter value or the gradient value, which are influenced by choice of activation function, initialization, and network architecture.

5 Experiments

We run experiments on MNIST. We use the same model for all experiments and fix the batch size to 500. Our model code is copied from the TensorFlow tutorial and contains 4 trainable layers: 2D Convolution, Max-pooling, 2D Convolution, Max-pooling, Fully-connected, Fully-connected. The convolutional layers contain trainable tensors with dimensions: (5, 5, 1, 32), (5, 5, 32, 64), and bias vectors. The Fully-connected layers contain trainable tensors with dimensions: (3136, 1024), (1024, 10), and bias vectors. The model is trained with Softmax loss, uses ReLu for hidden activations, and does not use BatchNorm.

---

1\text{size}(W) \text{ is the product of } W \text{'s dimensions. For a 2-D matrix, it is equal to } \# \text{ rows } \times \# \text{ columns, for a vector, it is equal to its length, etc. We also define size(scalar) } = 1.

2In practice, rather than converting \( \frac{a}{b} \) to \( \frac{a}{b+\epsilon \cdot \text{sign}(b)} \), we use \( \frac{a}{b+\epsilon} \cdot \text{sign}(b) \), as our goal is to push \( b \) slightly away from zero while keeping its sign.

3https://www.tensorflow.org/get_started/mnist/pros
5.1 MNIST Network Gradient Magnitudes

We record the gradient magnitudes for our 4-layer MNIST network throughout training. We compare the relative magnitude that $W$ changes, under vanilla SGD versus under PercentDelta. We plot $(||\Delta_{\text{SGD}}W||_1/||W||_1)$ and $(||\Delta_{\text{PercentDelta}}W||_1/||W||_1)$ for trainable tensors $W$ of convolutional and fully-connected layers. We notice that vanilla SGD proposes gradients that are not proportional to current weights. For example, using some learning rate, some layers can completely diverge (e.g. all entries switching signs) while other layers would change at a rate of <1%. In this case, to prevent divergence in any layer tensors, the learning rate would be lowered. However, using PercentDelta, relative magnitude of parameter updates is almost equal for all layers, showing that all layers are training at the same speed, consistently throughout the duration of training.

Figure 1: Magnitude of relative deltas for every layer, at training timestamps (batch at step 0, step 15, ...). Y-axis is the relative change of the L1 norm of every layer’s trainable tensor. For Stochastic Gradient Descent (SGD) and PercentDelta, respectively, we measure it as $(||\Delta_{\text{SGD}}W||_1/||W||_1)$ and $(||\Delta_{\text{PercentDelta}}W||_1/||W||_1)$. Disproportionate training is played down by the log-scale. Without log-scale, some SGD bars wont be visible. Curves are scaled by $\eta \gamma(t)$.

5.2 MNIST Test Accuracy Curves

We want to measure how fast can PercentDelta train MNIST. We compare training speed with other algorithms, including per-parameter adaptive learning rate algorithms, AdaGrad (Duchi et al., 2011) and Adam (Ba and Kingma, 2015), as well as a recent algorithm with similar spirit, LARS (You et al., 2017), which also normalizes the gradient w.r.t. a weight tensor, by the current value of the weight tensor.

Figure 5.2 In our experiments, we fix the learning rate of PercentDelta to 0.03, use momentum, and set $\gamma(t)$ to Equation 10. For Adagrad and Adam, we sweep the learning rate but we fix $\gamma(t) = 1$, as they implement their own decay. We do not use Momentum for Adagrad or Adam, as the later applies its own momentum scheme. For LARS, we use momentum, vary the learning rate, and set $\gamma(t)$ to Equation 10 with $m = 0.01$. We note the following:

1. In early stages, MNIST test accuracy climbs up the fastest with our algorithm.
2. The final test accuracy produced by our algorithm is higher, given the training budget of 5000 steps.

6 Discussion

6.1 Situations where PercentDelta is useful

While PercentDelta has outperforms other training algorithms on 4-layer MNIST, the space of models-datasets is enormous we leave it as future work to try PercentDelta under various models and datasets. Nonetheless, we speculate that PercentDelta (and similarly, LARS, You et al. (2017)) would be very useful in the following scenarios:

1. Learning Embeddings. Consider the common setup of feeding word embeddings (or graph embeddings, Abu-El-Haija et al., 2017) into a shared Neural Network and jointly learning the embeddings and Neural Network for an upstream objective. In this setup, a certain
Figure 2: MNIST experiments – Accuracy on Test set VS number of training steps. Each training step processes a batch size of 500 examples. The test accuracy is recorded after every 5-th steps. Figure contains subplots organized in 3 columns and 5 rows. **Columns** indicate the training stage. Left column shows early training (initial 200 steps) with y-axis starting at 0.5. Middle and right column show entire training curve, respectively showing raw values and smoothed values. **Rows** indicate algorithms. Each line color is consistent across the row and refers to the same hyper-parameter setting. We color the best performer with **red**. Top-to-bottom: PercentDelta (our algorithm), AdaGrad [Duchi et al., 2011], Adam [Ba and Kingma, 2015], LARS [You et al., 2017], and the last row shows the best performer from all algorithms.
embedding vector is only affected by a fraction of training examples, while the shared network parameters are affected by all training examples. The sum of gradients w.r.t. the shared network parameters over all training examples can be disproportionately larger than embedding gradients. PercentDelta ensures that the shared network is not being updated much faster than the embeddings.

2. Soft-Attention Models on Bag-of-Words. It is common to convert from variable-length bag-of-words \((x_j[1], x_j[2], \ldots)\) into fixed-length representation by a convex combination: \(x_{j+1} := \sum \alpha_i x_j[i]\), which can then be used for an upstream objective (e.g. event detection in videos, Ramanathan et al., 2016). Here, \(\alpha_i\) can be the \(i\)-th position of the softmax over all Words. The parameters of the softmax model would receive gradients from all words. PercentDelta ensures that, the otherwise disproportionately large, gradient updates of the softmax model are proportional to the remainder of the network.

3. Matrix Factorization Models. For example, Koren et al. (2009) propose to factorize a user-movie rating matrix \(R \in \mathbb{R}^{u \times m}\) into:
\[
R \approx W_U \times W_M + b_U \times \bar{1}^T + \bar{1} \times b_M + b,
\]
where \(W_U \in \mathbb{R}^{u \times d}\) and \(W_M \in \mathbb{R}^{d \times m}\) are the user and movie embedding matrix; \(d\) is the size of the latent-space; \(b_U \in \mathbb{R}^u\) and \(b_M \in \mathbb{R}^m\) are the user and movie bias vectors, and \(b \in \mathbb{R}\) is the global bias scalar. In this setup, \(b\) would receive very large sum-of-gradients, and PercentDelta can ensure that all parameters are training at the same speed.

6.2 Hyperparameters and Decay Function

It seems that PercentDelta has many knobs to tune. However, we can fix \(\eta\) to some value and only change \(\gamma(t)\) as their product determines the effective rate of change across all layers' trainable tensors. We can set \(\gamma(t)\) to constant decay:
\[
\gamma(t) = 1 - t \times m,
\]
where \(0 < m << 1\) determines the decay slope. In addition, we can ensure that \(\gamma(t) > 0\) to allow training continue indefinitely, by modifying Equation 9 to:
\[
\gamma(t) = \max(\beta, 1 - t \times m),
\]
where \(\beta\) can be set to a small positive value, such as 0.01. In this case, if we fix \(\eta = 0.03\), then we are effectively changing each trainable tensor by 3% for every training batch initially, then gradually annealing this change-rate to 0.03% after \(\frac{1}{m}\) steps.

More importantly, we feel that \(\eta\) and \(\gamma(t)\) are a function of the dataset, and not the model. Experimentally, we observe the algorithm is insensitive to the choices of \(m\) and \(\eta\) as long as they are “reasonable” (i.e. removing diverging setups that can be quickly detected). However, we do not yet have a formula to automatically set them. Nonetheless, with a wide range of \(\eta\) and \(m\), we experimentally show on MNIST that PercentDelta beats all training algorithms, given the same budget of training steps.

7 Conclusion

We propose an algorithm that trains layers of a neural network, all at the same speed. Our algorithm, PercentDelta, is a simple modification over standard Gradient Descent. It divides the gradient w.r.t. a trainable tensor over the mean of \(||\text{gradient / tensor}||_1\). The division over mean L1-norm is scalar, and only changes the gradient’s magnitude but not its direction. Effectively, this updates the L1 norm of trainable layers, all at the same rate. We recommend a linear decaying change-rate schedule. Our modified gradients can be passed through a standard momentum accumulator (Sutskever et al., 2013).

Overall, we show experimentally that our algorithm puts an upper envelop on all training algorithms, reaching higher test accuracy with fewer steps.

References

Abu-El-Haija, S., Perozzi, B., and Al-Rfou, R. (2017). Learning edge representations via low-rank asymmetric projections. In ACM International Conference on Information and Knowledge Management (CIKM).
Ba, J. and Kingma, D. (2015). Adam: A method for stochastic optimization. In *International Conference on Learning Representations*.

Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization. In *arxiv*.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. In *Journal of Machine Learning Research*.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS ’10)*.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Journal of Machine Learning Research (JMLR)*.

Koren, Y., Bell, R. M., and Volinsky, C. (2009). Matrix factorization techniques for recommender systems. In *IEEE Computer*.

Ramanathan, V., Huang, J., Abu-El-Haija, S., Gorban, A., Murphy, K., and Fei-Fei, L. (2016). Detecting events and key actors in multi-person videos. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. In *Nature*.

Salimans, T. and Kingma, D. P. (2016). Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Neural Information Processing Systems*.

Sutskever, I., Martens, J., Dahl, G. E., and Hinton, G. E. (2013). On the importance of initialization and momentum in deep learning. In *International Conference on Machine Learning*.

You, Y., Gitman, I., and Ginsburg, B. (2017). Scaling sgd batch size to 32k for imagenet training. In *UC Berkeley Technical Report*. 