Tricks from Deep Learning[*]

Atılım Güneş Baydin[†] Barak A. Pearlmutter[‡] Jeffrey Mark Siskind[§]

April 2016

Introduction

The deep learning [1, 2, 3] community has devised a diverse set of methods to make gradient optimization, using large datasets, of large and highly complex models with deeply cascaded nonlinearities, practical. Taken as a whole, these methods constitute a breakthrough, allowing computational structures which are quite wide, very deep, and with an enormous number and variety of free parameters to be effectively optimized. The result now dominates much of practical machine learning, with applications in machine translation, computer vision, and speech recognition. Many of these methods, viewed through the lens of algorithmic differentiation (AD), can be seen as either addressing issues with the gradient itself, or finding ways of achieving increased efficiency using tricks that are AD-related, but not provided by current AD systems.

The goal of this paper is to explain not just those methods of most relevance to AD, but also the technical constraints and mindset which led to their discovery. After explaining this context, we present a "laundry list" of methods developed by the deep learning community. Two of these are discussed in further mathematical detail: a way to dramatically reduce the size of the tape when performing reverse-mode AD on a (theoretically) time-reversible process like an ODE integrator; and a new mathematical insight that allows for the implementation of a stochastic Newton's method.

The Deep Learning Mindset

The differences in mindset between the AD and deep learning communities are rooted in their different goals and consequent practices. To grossly caricature the situation, the communities have different typical workflows.

A Typical AD Workflow

1. Primal computation (e.g., climate simulation) is given.
2. Calculate exact derivatives (e.g., the gradient) automatically and efficiently.
3. Use these derivatives (e.g., for sensitivity analysis or in a standard gradient optimization method).
4. Iterate to improve end-to-end accuracy throughout.

A Typical Deep Learning Workflow

1. Construct primal process whose derivatives are "nice", meaning easy to calculate and to use for optimization.
2. Manually code approximate (e.g., stochastic) gradient calculation.
3. Use in custom manually-assisted stochastic gradient optimization method.
4. Iterate to improve generalization on novel data.

Given these different settings, it is understandable that the deep learning community has developed methods to address two problems.

(I) **Methods for making a forward process whose gradient is "nice" in an appropriate sense.**

The important difference here from the usual situation in AD is that, in the deep learning community, the "primal" computational process being trained needs to have two properties: it needs to be sufficiently powerful to perform the desired complex nonlinear computation; and it has to be possible to use training data to set this primal processes' parameters to values that will cause it to perform the desired computation. It is important to note that this parameter setting need not be unique.

When using gradient methods to set these parameters, there are two dangers. One is that the system will be poorly conditioned. An intuition for stiffness in this context is that, in a stiff system, changing one parameter requires precise compensatory changes to other parameters to avoid large increases in the optimization criterion. The second danger is that the gradient will be uselessly small for some parameters. Such small gradients would be expected in deeply nested computational process relating inputs to outputs. After all, each stage of processing has a Jacobian. These Jacobians have spectra of singular values. If these singular values are all less than one, then the gradient will be washed out, layer by layer, during the backwards sweep of reverse AD. And if the singular values exceed one, the gradients will instead grow exponentially, leading to a similar sort of catastrophe.

In response to these difficulties, the deep learning community has come up with a variety of architectural features which are added to the primal process to give it a better behaved gradient, and has also come up with

[*] **Extended abstract presented at the AD 2016 Conference, Sep 2016, Oxford UK.**

[†] Corresponding Author, Dept of Computer Science, National University of Ireland Maynooth, gunes@cs.nuim.ie (Current address: Dept of Engineering Science, University of Oxford, gunes@robots.ox.ac.uk)

[‡] Dept of Computer Science, National University of Ireland Maynooth, barak@pearlmutter.net

[§] School of Electrical and Computer Engineering, Purdue University, qobi@purdue.edu

modifications of the reverse AD procedure resulting in the calculation of what we might call a pseudo-gradient, which is designed to be more useful than the true gradient for purposes of optimization.

(II) **Faster and more robust stochastic gradient optimization methods.**

Simply calculating the actual gradient is so slow in deep learning systems that it is impractical to use the true gradient for purposes of optimization. Instead much faster computations are used to calculate an (unbiased) estimate of the gradient, and these are used for optimization. Such *stochastic optimization* [4] were first proposed at the dawn of the computer age, but have in practice been something of a black art, requiring constant manual intervention to achieve acceptable performance. This unfortunate state of affairs has now been rectified [5], with the development of algorithms that are asymptotic improvements over their classical antecedents, as well as methods for automatically controlling the stochastic optimization process not just in its terminal asymptotic quadratic phase (which is in practice of little interest in deep learning systems) but also during the so-called initial transient phase.

Deep Learning Methods

We proceed to discuss methods developed by the machine learning community to address each of these issues, in turn. First, *gradient tricks*, namely methods to make the gradient either easier to calculate or to give it more desirable properties. And second, *optimization tricks*, namely new methods related to stochastic optimization.

Gradient Tricks

A naïve description of the basic setup in deep learning would be a many-layers-deep network of perceptrons, that is trained by computing the gradient of the training error through reverse mode AD [6], usually with some layers of linear convolutions [7, 8] and occasionally recurrent structures [9, 2]. The reality, however, is not so simple, as such systems would exhibit a number of problems.

- Standard gradient descent with random initial weights would perform poorly with deep neural networks [10].
- Reverse AD blindly applied to the forward model would be highly inefficient with regard to space.
- Reverse AD, in most implementations, would unroll the matrix operations of the forward model rendering the adjoint calculations quite inefficient.
- The system would be very poorly conditioned, making standard optimization methods ineffective.
- Parameters would vary wildly in the magnitude of their gradient, making standard optimization methods ineffective, and perhaps even causing floating point underflow or overflow.

Starting with the breakthrough of layer-by-layer unsupervised pretraining (using mainly restricted Boltzmann machines) [11], the deep learning community has been developing many methods to make gradient descent work with deep architectures. These methods (related to model architectures and gradient calculation) are so numerous, that we cannot hope to even survey the current state of the art in anything shorter than a book. Here we discuss only a small sample of methods. Even these benefit from having some structure imposed by dividing them into categories. These categories are rough, with many methods spanning between a few.

Efficiency

Care is taken to fully utilize the available computational resources, and to choose appropriate tradeoffs between speed, storage, and accuracy.

(a) *Commuting ∇ with $\sum$.* The gradient operator is linear, and therefore commutes with sum, or equivalently with the average which we will denote with angle brackets $\langle\cdot\rangle$. In other words, $\nabla_w \langle E(x_p; w)\rangle_p = \langle \nabla_w E(x_p; w)\rangle_p$, where $E(x_p; w)$ is the error of the network with parameters w on training pattern p. If the left hand side of this equation were coded and the gradient calculated with any AD system using reverse AD, the result would be highly inefficient, as the "forward sweep" would encompass the entire training set, and would consume enormous storage and would be hard to parallelize. The second form, in contrast, lends itself to parallelism across training cases, and each "little" gradient requires a constant and very manageable amount of storage.

(b) *Stochastic Gradient.* It is not really feasible to calculate the true gradient when there is a large dataset, as just calculating the true error $E = \langle E_p \rangle = \frac{1}{|D|}\sum_{p \in D} E_p$ over all patterns p in the dataset D is impractically slow. Instead a sample is used. Calculating the gradient of the error of just a random sample from the dataset (or sequential windows, assuming the dataset has been randomly shuffled) is much faster. The expectation of that process is the true value, so this is admissible for use in stochastic gradient optimization.

(c) *GPUs, and low-precision numbers.* Most deep learning work uses GPUs, typically very large clusters of GPUs. 32-, instead of 64-bit, precision floating point is common; and fixed point, with 16-bit or even less precision, is feasible due to the error resiliency of neural networks [12]. No current AD systems can manage the delicate tradeoffs concerning which quantities require more or less precision in their representations. Note that derivatives often need more precision, or have a different numeric range, than the primal quantities they are associated with.

(d) *Mini-Batches and data parallelism.* In calculating the stochastic gradient, it is tempting to do the minimal amount of computation necessary to obtain an unbiased estimate, which would involve a single sample from the training set. In practice it has proven much better to use a block of contiguous samples, on the order of dozens. So instead of E_p for some p, one uses $n^{-1}\sum_{i=0}^{n-1} E_{p+i}$ for some p. This has two advantages: the first is less noise, and the second is that data-parallelism can be used, with favorable cache properties where every quantity in the network is replaced by an n-vector, utilizing SIMD on CPUs and GPUs. Conventional AD systems could not maintain this sort of allocation-free vector parallelism through the reverse AD transform.

(e) *Reversible learning.* In deep learning it is sometimes desired to perform bi-level optimization, usually in order to tune hyperparameters of the training process. Naïvely, since the primal process is $w(t+1) = w(t) - \eta \nabla_w E(x_t; w(t))$, this would require saving all the $w(1), \ldots, w(t), w(t+1), \ldots, w(T)$. Since $w(t)$ may consist of millions of parameters, and T will typically be in the millions, this is not really feasible. Fortunately a clever technique has been developed to do reverse AD through this process with less storage [13].[1] The trick is to note that this process looks suspiciously like integration of a time-reversible ODE. So we could try to run the system backwards while in the reverse AD reverse pass, calculating necessary quantities as needed: $w(t) \approx w(t+1) + \eta \nabla_w E(x_t; w(t+1))$. The would be approximate due to floating point inaccuracy as well as the use of $w(t+1)$ rather than $w(t)$ in $E(\cdot)$. However these inaccuracies would typically be very small, so the difference between these could be computed during the *forward* sweep, and encoded efficiently in a highly compressed form.

This same technique seems also applicable to performing reverse AD through any process which is (theoretically) time-reversible, such as most ODE or PDE integrators. Of course, the savings here amount to only a constant factor (albeit perhaps a very large one) over standard reverse AD. Combining the method with checkpoint reverse would seem natural, and would allow much larger leaf nodes in the checkpoint AD reverse computation graph.

Vanishing or Exploding Gradients

In a multi-layered structure, one would expect the gradients of quantities at early layers to be nearly zero (assuming the gains at intermediate levels are below unity) or to be enormous (assuming the gains are above unity). Some methods avoid this. Others work around it, effectively doing a block-structured diagonal pre-conditioning of the gradient.

(a) *Rectified linear units instead of sigmoids.* Classic multi-layer perceptrons use the sigmoid transfer function $\xi \mapsto 1/(1 + \exp(-\xi))$, but this has a derivative which goes to zero when $\xi \gg 0$. That means that when a unit in the network receives a very strong signal, it becomes difficult to change. Using a rectified linear unit (ReLU) transfer function, $\xi \mapsto \max(0, \xi)$ overcomes this problem, making the system more plastic even when strong signals are present. Many variants of this have been proposed, often to avoid the lack of continuity in the derivative of the ReLU. One such are exponential linear units (ELUs) [14].

(b) *Long short-term memory (LSTM).* A recurrent network, unfolded in time, is simply a deep network with some invariants imposed on the parameter matrices. One technique that has proven useful in allowing gradient information to span long temporal spans in the context of recurrent networks, or equivalently many layers in a deep network, is the LSTM architecture [15]. This is essentially a hold-value unit which can have quantities gated in and out. The recent gated recurrent unit (GRU) model [16] is a simplification of this idea.

(c) *Gradient clipping.* In the domain of deep learning, there are often outliers in the training set: exemplars that are being classified incorrectly, for example, or improper images in a visual classification task, or mislabeled examples, and the like. These can cause a large gradient inside a single mini-batch, which washes out the more appropriate signals. For this reason a technique called gradient clipping [17] is often used, in which components of the gradient exceeding a threshold (in absolute value) are pushed down to that threshold.

Conditioning

Keeping the error surface well conditioned for gradient optimization has been one of the keys to the current widespread deployment of deep learning.

(a) *Dropout.* Imagine a network in which multiple units together represent some important feature, requiring a precise weighting of their values in downstream processing. This would make optimization quite difficult, as it sets up couplings between parameters which must be maintained. A technique to avoid such "unfortunate collusions" is dropout [18], in which each unit in the network is, on each training pattern pass, randomly "dropped out" with some probability (typically 50%) by holding its value at zero. This encourages detected features to be independently meaningful. (In "production mode" dropout is turned off, and the weights scaled to compensate, to minimize the noise when performance matters.)

(b) *Careful initialization.* Considering how the variances of activation values and gradients can be maintained between the layers in a network leads to intelligent normalized initialization schemes, which enable substantially faster optimization convergence [10].

Optimization Tricks

(a) *Early stopping.* When fitting a dynamic system to data, as exact a match as possible is desired, so the true optimum is sought. This is not the case in machine learning, where the optimization is of error on a training set, while the primary concern is generally not performance on the training set, but on as-yet-unseen new data. There is often a tradeoff between the two, encountered after optimization has proceeded for a significant amount of time. This is addressed by early stopping [19], in which an estimate of performance on unseen data is maintained, and optimization is halted early when this estimated generalization performance stops improving, even if performance on the training set is continuing to improve. (An estimate of generalization is often obtained by holding back some training data and using it only for this purpose, and never for optimization.)

[1] Another possibility, not used in the deep learning community, would be checkpoint reverse mode.

(b) Many AD practitioners have had the demotivating experience of explaining how AD can be used to efficiently calculate Hessian-vector products, only to be asked whether it might be possible to instead calculate the Hessian-inverse-vector product. Although an efficient way to perform this calculation exactly remains unknown, a method has been discovered to efficiently calculate an *unbiased estimate* of the Hessian-inverse-vector product [20]! Note that we can express a matrix inverse as a series $H^{-1} = \sum_{i=0}^{\infty}(I - H)^i$ where we assume that the spectrum of H allows convergence. We could obtain an unbiased estimate of this sum as follows. Let $p(i)$ be a distribution with support for all integers $i \geq 0$. Choose $i \sim p(i)$. Calculate $p(i)^{-1}(I - H)^i$. This is equal, in expectation, to H^{-1}. Similarly $p(i)^{-1}\overbrace{(I-H)(\cdots((I-H)}^{i \text{ times}}v))$ is an unbiased estimate of $H^{-1}v$, and can be computed with i Hessian-vector products. If $p(\cdot)$ is chosen to have small mean, then i will usually be small. If instead of an exact Hessian-vector product we can instead only compute an unbiased estimate of this product, the same procedure will work, except that each H in the above expression becomes $\hat{H}$, an operator that performs a stochastic unbiased Hessian-vector product. Further development improves the efficiency of the computation, and embeds it in a stochastic Newton's method with proven efficiency properties.

Conclusion

Many advances in AD, both longstanding and recent, are of great potential utility to machine learning in general and deep learning in particular. Analogously, the machine learning community in general, and the deep learning community in particular, have been using computational derivatives "in anger" for quite some time. They have been forced to build very large systems whose optimization would be intractable using generic AD systems and batch gradient optimization. Necessity has been the mother of invention, and they have discovered a variety of novel methods which allow them to handle large systems and enormous datasets. Many of these methods are related to AD in some fashion, and it is our hope that the AD community will find them of interest.

Acknowledgments

This work was supported, in part, by Science Foundation Ireland grant 09/IN.1/I2637 and by NSF grant 1522954-IIS. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[2] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016. URL http://www.deeplearningbook.org.

[4] Herbert Robbins and Sutton Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22(3):400–407, 1951. doi: 10.1214/aoms/1177729586.

[5] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT'2010)*, pages 177–187, Paris, France, 2010. Springer.

[6] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:9, 1986.

[7] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, 1980. ISSN 1432-0770. doi: 10.1007/BF00344251.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[9] Barak A. Pearlmutter. Gradient calculations for dynamic recurrent neural networks: A survey. *IEEE Transactions on Neural Networks*, 6(5):1212–1228, 1995.

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

[11] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[12] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. *arXiv:1502.02551*, 2015.

[13] Dougal Maclaurin, David Duvenaud, and Ryan P. Adams. Gradient-based hyperparameter optimization through reversible learning. *arXiv:1502.03492*, 2015.

[14] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (ELUs). *arXiv:1511.07289*, 2015.

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

[16] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv:1412.3555*, 2014.

[17] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of The 30th International Conference on Machine Learning*, pages 1310–1318, 2013.

[18] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[19] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural Networks: Tricks of the Trade*, pages 437–478. Springer, 2012.

[20] Naman Agarwal, Brian Bullins, and Elad Hazan. Second order stochastic optimization in linear time. *arXiv:1602.03943*, 2016.