Stochastic Gradient Descent Algorithm in the Computational Network Toolkit

Brian Guenter, Dong Yu, Adam Eversole, Oleksii Kuchaiev, Michael L. Seltzer
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
{bguenter, dongyu, adame, olekku, mseltzer}@microsoft.com

Abstract

We introduce the stochastic gradient descent algorithm used in the computational network toolkit (CNTK) — a general purpose machine learning toolkit written in C++ for training and using models that can be expressed as a computational network. We describe the algorithm used to compute the gradients automatically for a given network. We also propose a low-cost automatic learning rate selection algorithm and demonstrate that it works well in practice.

1 Computational Network Toolkit

A computational network (CN) is a directed graph in which each leaf represents an input value or a learnable parameter and each node represents an operator. Figure 1 illustrates an example CN of a log-linear model. Here, each node is identified by a {node name : operator type} pair and takes its ordered children as the operator's inputs. For example, in the figure, $\mathbf{T} = Times(\mathbf{W}, \mathbf{X})$ which is different from $\mathbf{T} = Times(\mathbf{X}, \mathbf{W})$. A CN can have many root nodes which are used under different conditions. For example, one root node may represent a cross-entropy training criterion and another may represent an evaluation criterion. The network in Figure 1 has only one root node {$\mathbf{C}$: Cross Entropy}. Many machine learning models, such as neural networks, that can be described via a series of operations, can be converted into a CN.

The computational network toolkit (CNTK) is a general purpose C++ based machine learning toolkit for models that can be described as CNs. Figure 2 illustrates the architecture of CNTK. The core of CNTK is an internal representation of a CN which provides two key methods: *Evaluate*, which computes the value of a node given its inputs and *Compute Gradient*, which computes the gradient of a node with respect to its inputs. These methods are executed using an *IExecutionEngine* such as a CPU, a GPU, or a data flow graph such as pTask [1]. *ICNBuilder* reads the network description (or language) and creates a CN. *IDataReader* reads in features and labels stored in different formats. *ILearner* optimizes the model parameters with different optimization algorithms. In this paper, we focus on the well-known stochastic gradient descent (SGD) algorithm. Section 2 describes how gradients are computed efficiently regardless of the CN architecture. In Section 3 a new learning rate selection algorithm is proposed and evaluated. Section 4 provides some practical tricks used in the toolkit's implementation.

2 Automatic Gradient Computation

The objective function, f, for training the computational network is of the form $f : \mathbb{R}^m \rightarrow \mathbb{R}^1$. The gradient of this function is

$$D(f(g_1(h_1, \ldots, h_m), \ldots, g_n(\ldots)) = \sum_{i=1}^{n} \frac{\partial f}{\partial g_i} D(g_i) \tag{1}$$

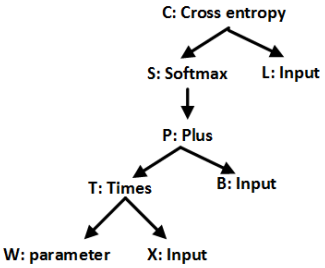

Figure 1: A log-linear model represented as a computational network

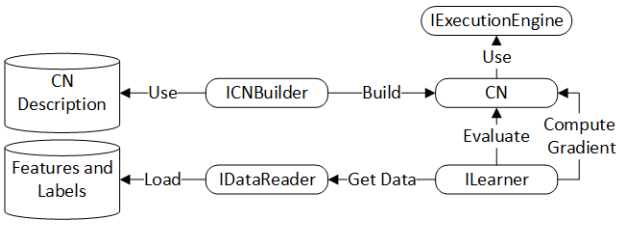

Figure 2: Architecture of CNTK.

where the $g_i(\dots)$ are themselves functions of some h_j and so on. Naively applying this recursion to the function graph will lead to exponential time worst-case behavior as graph nodes will be evaluated multiple times.

Factoring out the common prefix terms $\frac{\partial f}{\partial g_i}$ makes computation linear in the number of computation nodes in the graph. This is analogous to common subexpression elimination in a conventional expression graph, only here the common subexpressions are the *parents* of the nodes, rather than the children. For scalar functions this factorization can be performed with a simple recursive algorithm[2]. However, for matrix expressions we need differentiation rules that express partial derivatives in terms of matrix expressions, rather than as a large scalar function expression graph.

For functions applied per matrix element, such as sigmoid, the differentiation rules are analogous to the scalar case. Matrix addition is similarly straightforward. The only case that requires some consideration is matrix multiplication. Suppose our objective function is $f(\mathbf{C}) = \|\mathbf{C}\|$ where $\mathbf{C} = \mathbf{AB}$ and $\|\mathbf{C}\|$ is the Frobenius norm of $\mathbf{C}$. Writing the elements of the gradient of f with respect to $\mathbf{A}$

$$\frac{\partial f}{\partial a_{k,l}} = \sum_i \sum_j \frac{\partial f}{\partial c_{i,j}} \frac{\partial c_{i,j}}{\partial a_{k,l}} \tag{2}$$

and noticing that

$$\frac{\partial c_{i,j}}{\partial a_{k,l}} = \begin{cases} b_{lj} & i = k \\ 0 & \text{otherwise} \end{cases}$$

we can rewrite eq. 2 as

$$\frac{\partial f}{\partial a_{k,l}} = \sum_j \frac{\partial f}{\partial c_{k,j}} b_{l,j} \tag{3}$$

If we define matrix $d\mathbf{C}$ with elements

$$d\mathbf{C}_{i,j} = \frac{\partial f}{\partial c_{i,j}}$$

then we can write eq. 3 as

$$d\mathbf{A}_{k,l} = \sum_j d\mathbf{C}_{k,j} b_{l,j}. \tag{4}$$

Eq. 4 is just the formula for the matrix multiplication $d\mathbf{A} = d\mathbf{C}\mathbf{B}^\mathbf{T}$. A similar derivation shows that $d\mathbf{B} = \mathbf{A}^\mathbf{T} d\mathbf{C}$. A similar approach was used in Theano [3] although the implementation is different.

A simple recursive algorithm for computing the gradient is shown in Algorithm 1. The function is called on the root node of the computation network, with dC argument set to a 1x1 matrix containing the value 1. Each node type, such as *, +, or sigmoid, defines its own $node.d(child)$ function.

3 Automatic Learning Rate Selection

The performance of the stochastic gradient descent (SGD) algorithm depends critically on how the learning rates are chosen. Many researchers have proposed schemes for making learning rates adaptive [4]. These methods typically can be classified into two categories: those based on an estimated

Algorithm 1 Recursive Gradient

function GRAD=GRADIENT(f, dC)
Input: f=root node of computational network, dC=derivative of parent node
Output: df=new network representing the derivative of f
if $timesVisited$ = $parentCount$ **then**
foreach child of this $node$ grad$(child, node.d(child))$
else
$node.derivative$ += dC
$timesVisited+ = 1$
end if
end function

Hessian and those based on an empirical search algorithm. We have found these existing schemes to be expensive when applied to large datasets. For example, the efficient Hessian estimation algorithm used in [5] incurs one additional back-propagation pass or 100% additional computation cost. A full range empirical search typically requires processing the training data with different learning rates at least 10 times, even if an efficient algorithm is used to adjust the learning rate based on the sign of improvement. Such a large expense is not affordable for real-world tasks with millions or billions of training samples.

We propose a novel method for automatically adjusting learning rates. Similar to that proposed in [5], our approach can increase or decrease the learning rate and is suitable for non-stationary problems. Unlike other empirical methods, however, our algorithm determines the learning rate before each epoch using a small sample of the training set. This requires only minimal additional computation. Algorithm 2 describes the steps of the learning rate search algorithm. During the first epoch, in which the model parameters have typically been randomly initialized, the algorithm searches for the best learning rate based on a set of n random samples using a grid search which decreases the learning rate by 0.618 each time (0.5 works equally well). For all subsequent epochs, the algorithm searches for the largest learning rate that is sufficient to see improvement in the average training criterion if all N samples in the epoch are used to update the parameters. Note that when the learning rate is fixed, the earlier samples typically improve the training criterion more than the later samples since the error signal becomes smaller over time. We found that we can approximate this behavior by assuming that the gain is proportional to the square root of the number of samples seen. The learning rate selection algorithm is typically run on 2-5% (or 200-500 mini-batches) of the full epoch size.

Note that starting with the second epoch, there are actually two alternative criteria to select the learning rates: one is to select the best learning rate just as in the first epoch, and one is to select the largest learning rate that improves over the 0 learning rate. In practice, we have found that these two approaches will reduce the learning rate quickly and lead to an inferior final model.

Algorithm 2 Learning Rate Search

function λ=SEARCHLEARNINGRATE(N, n, e_{t-1})
Input: N=total number of samples in an epoch; n=number of samples used to estimate learning rate; e_{t-1}=training criterion from previous epoch
Output: λ=optimal learning rate
if $t \leq 0$ **then**
λ=Grid search the best λ to minimize e^s_λ
else
Compute e^s_0 with learning rate 0 by running SGD over n samples
$r = sqrt(n/N)$
$e_{base} = (1 - r) \times e^s_0 + r \times e_{t-1}$
λ=Grid search for the largest λ so that $e^s_\lambda \leq e_{base}$
end ifreturn λ
end function

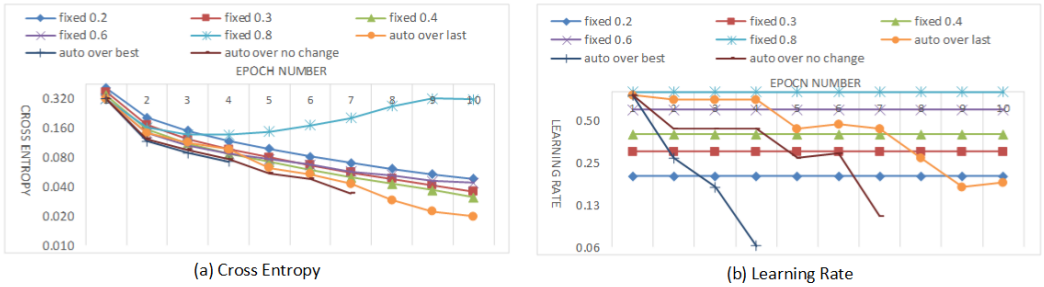

Figure 3: Comparison of different learning rate adjustment strategies on MNIST.

Figure 3 (a) and (b) illustrate the training set cross-entropy improvement and the learning rate selected over epochs with different learning rate adjustment schemes on the MNIST data set [6]. In these experiments, we have used a neural network with 784 input units, 256 hidden units, and 10 output units. The SGD algorithm used a mini-batch size of 32. The total training set has 60,000 samples, of which 3200 samples (or 5%) were used to adjust the learning rate. On average each epoch required 3 passes through the 3200 samples which increased the total training time by 15%. In the figure, we can clearly see the inferior performance of other two alternatives. We can also observe that the proposed approach performs better than the fixed-learning rate search approach in training set performance. Note that the proposed algorithm may increase the learning rate.

We have applied this algorithm to the speech data and advertisement click prediction data and outperformed a hand-tuned learning rate in both cases. For example, on the Switchboard 24-hour training set with 1502 classes (tied triphone states), the proposed algorithm, which used 1.5% samples and 5% overhead in learning rate search, together with dropout [7] at a rate of 0.2 to control overfitting, achieved 49.1% frame accuracy on the evaluation set without pretraining, compared to 47.3% accuracy achieved by the best hand-tuned learning rate strategy with pretraining.

4 Practical Tricks

Most of the work performed in *Evaluate* and *Compute Gradient* functions is expressed using matrix operations. Currently, CNTK implements matrix operation using the following BLAS libraries: ACML math library from AMD and cuBLAS from NVIDIA, for CPU and GPU computations, respectively. Since cuBLAS doesn't implement all the functions necessary for CN, we had to implement many element-wise matrix operations and other functions using custom CUDA kernels. One of the most important considerations when optimizing GPU code is minimizing memory copies between CPU RAM and GPU RAM. For example, when computing training loss per minibatch the resulting value is kept on the GPU rather than transferring it to CPU memory to avoid either synchronization or asynchronous memory copies. We also set cuBLAS pointer mode to CUBLAS_POINTER_MODE_DEVICE which allows it to keep call results in GPU memory.

When implementing CUDA kernels, making use of shared memory and achieving higher levels of parallelism is necessary for a good performance. Functions like *Max()*,*Sum()*, *Frobenius norm()*, etc. are implemented using a reduction-based approach maximizing the use of shared memory within each thread block. The goal is to split the work among as many threads as possible since thread overhead is negligible for CUDA. For classification tasks with many output classes (such as 1502 tied triphone states), efficiency of functions such as *AssignColumnwiseSoftmax()* is important. In our implementation, we assign a separate CUDA block with many threads to each matrix column and use a reduction-based approach several times within each column to achieve greater speed.

5 Summary

In this paper we described the SGD algorithm used in the CNTK with a focus on the automatic gradient computation and the learning rate adjustment algorithm. We confirmed the effectiveness of the proposed approach in several real-world tasks.

References

[1] C. J. Rossbach, J. J. Currey, M. Silberstein, B. Ray, and E. Witchel, "PTask: Operating system abstractions to manage GPUs as compute devices," in *Symposium on Operating Systems Principles*, 2011.

[2] Brian Guenter, "Efficient symbolic differentiation for graphics applications," in *ACM SIGGRAPH 2007 papers*, New York, NY, USA, 2007, SIGGRAPH '07, ACM.

[3] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio, "Theano: a CPU and GPU math expression compiler," in *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010, Oral Presentation.

[4] Abraham P George and Warren B Powell, "Adaptive stepsizes for recursive estimation with applications in approximate dynamic programming," *Machine learning*, vol. 65, no. 1, pp. 167–198, 2006.

[5] Tom Schaul, Sixin Zhang, and Yann LeCun, "No more pesky learning rates," *arXiv preprint arXiv:1206.1106*, 2012.

[6] Yann LeCun and Corinna Cortes, "The mnist database of handwritten digits," 1998.

[7] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," http://arxiv.org/abs/1207.0580, 2012.