A Generic Dataflow Programming Environment for Neural Networks

Technical Report IDSIA-15-00
February 4, 2000

Nicol N. Schraudolph
nic@idsia.ch

IDSIA, Galleria 2, 6928 Manno, Switzerland
http://www.idsia.ch/

Introduction. Any computation can be visualized as a *dataflow* — a directed graph along whose edges data flows, to be computed upon at the nodes. *Dataflow programming* is a form of visual programming in which the user specifies the dataflow explicitly by drawing its graph; well-known examples include the `Labview` and `Khoros` programming environments. The architecture of a neural network (in the widest sense) is in essence a condensed specification of its dataflow. Since network architecture must routinely be manipulated by the user as part of the model construction and selection process, dataflow programming is eminently suited for neural network development.

Data typically flows through a neural network in both directions: as activation flows forward, an error correcting feedback signal flows back. Some learning algorithms also require additional dataflow (such as second-order information) through the same network structure. The network architecture, then, is a *folded* description of the dataflow. Standard dataflow programming environments unfortunately do not support such folded descriptions. Some neural network simulators do, thus permitting direct, graphical manipulation of network architecture. The operators (nodes in the dataflow graph) they support, however, are rather coarse-grained and special-purpose — and entire Kohonen map or MLP layer, for instance. Furthermore, their use of the dataflow paradigm does not extend to other aspects of the computation (such as the learning algorithm), which are thus far less accessible to the user.

We are developing a generic dataflow programming environment that supports folded dataflows, implements fine-grained operators (along the lines of `Matlab` functions), and handles all computation (including the learning algorithm) within the dataflow paradigm. We find that this system provides an excellent platform for the design of neural network algorithms and architectures.

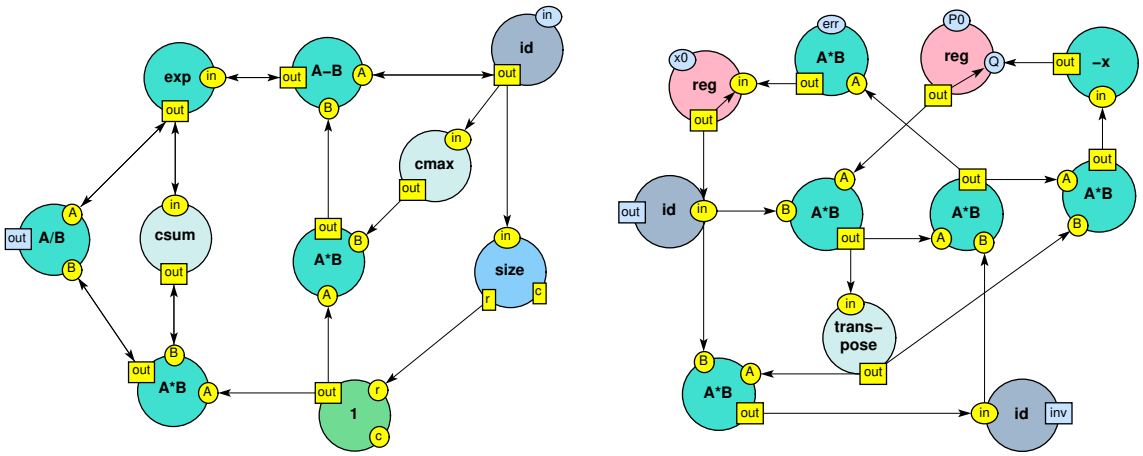
Modules. Modules are the nodes in our folded dataflow; they have one or more *ports* through which they connect to each other, to send and receive data matrices of various types. The matrix multiplication module, for instance, has two ports for its operands, and one for the result. Ports are generally bidirectional, since for most types of data there is an associated type flowing in the opposite direction. At present we support two data types: the default value/gradient data, and a directed derivative type that is used in the efficient computation of Hessian-vector products (Pearlmutter, 1994). Modules handle each data type in each direction; matrix multiplication thus comprises the following six matrix operations:

direction	value/gradient	Hessian-vector
$A, B \to C$	$C = A\,B$	$\mathcal{R}(C) = \mathcal{R}(A)\,B + A\,\mathcal{R}(B)$
$A, C \to B$	$\frac{\partial}{\partial B} = A^T \frac{\partial}{\partial C}$	$\mathcal{R}(\frac{\partial}{\partial B}) = A^T \mathcal{R}(\frac{\partial}{\partial C}) + \mathcal{R}(A)^T \frac{\partial}{\partial C}$
$B, C \to A$	$\frac{\partial}{\partial A} = \frac{\partial}{\partial C} B^T$	$\mathcal{R}(\frac{\partial}{\partial A}) = \mathcal{R}(\frac{\partial}{\partial C})\,B^T + \frac{\partial}{\partial C}\,\mathcal{R}(B)^T$

The advantage of bundling these operations within a single object is that in such a system, once the network architecture is specified, the computation of derivative quantities (gradients, Hessian-vector products) is *guaranteed* to be correct. This removes a large source of error in neural network implementations.

In addition to algebraic operations, we have modules for file and screen I/O, data generation and visualization, as well as registers and control structures. In the underlying C++, modules are implemented as a class hierarchy, making the derivation of new modules a relatively simple affair.

Procedures. We provide hierarchical procedural abstraction by allowing dataflow graphs to be encapsulated in *procedures*. Procedures are a type of module, and can be used as such in the construction of other dataflow graphs (procedures). Programming in our system typically proceeds by building and elaborating a hierarchical library of appropriate procedures. Procedures are used to specify not only the network architecture, but also the learning algorithm; our library of procedures thus comprises procedures implementing, *e.g.*, stochastic gradient descent, matrix momentum (Orr, 1995), SMD (Schraudolph, 1999; Schraudolph & Giannakopoulos, 2000), and Kalman filtering, as well as typical neural network building blocks such as MLP layers, activation functions, and loss functions. To illustrate, here are the dataflow graphs of two procedures in our library — a softmax activation function (left), and an extended Kalman filter (right):



Compute Engine. One might think that the graphical nature of our programming environment, and its use of such fine-grained operators, would lead to rather inefficient execution. We are pleased to report that on the contrary, our dataflow compute engine is highly efficient. Data is cached at each port, and the overhead associated with the fine-grained modular structure has been all but eliminated by inlining all function calls handling communication between ports. One remaining inefficiency is some unnecessary memory allocation and data copying due to an overzealous caching policy.

By reversing its edges, our compute engine converts the dataflow graph to a *call graph* before executing it. This means that computation occurs strictly on demand — unless a datum is used in some way, it will never be computed. The resulting practical advantage is that all kinds of optional monitoring and debugging facilities may be provided in a procedure without slowing down its normal operation.

Graphical User Interface. Although our system can display a procedure's dataflow graph visually (as shown above), it does not yet have a truly interactive GUI — for the time being, dataflow graphs must be entered as lists of nodes and vertices. This impedes the effective use of our system for programming, debugging, and running neural network simulations. We intend to add a GUI that supports interactive construction, monitored execution, and visual debugging of dataflow graphs. We expect that with the added functionality of the GUI, our system will become an excellent prototyping environment for neural network research and development.

Key References:

Orr, G. B. (1995). *Dynamics and Algorithms for Stochastic Learning*. Ph.D. thesis, Oregon Graduate Institute.

Pearlmutter, B. A. (1994). Fast exact multiplication by the Hessian. *Neural Computation*, *6*(1), 147–160.

Schraudolph, N. N. (1999). Local gain adaptation in stochastic gradient descent. In *Proc. 9th Intl. Conf. Artificial Neural Networks (ICANN)*, pp. 569–574 Edinburgh, Scotland. IEE, London.

Schraudolph, N. N., & Giannakopoulos, X. (2000). Online independent component analysis with local learning rate adaptation. In *Advances in Neural Information Processing Systems 12*, pp. 789–795. The MIT Press, Cambridge, MA.