Treating Machine Learning Algorithms As Declaratively Specified Circuits

Jason Eisner Nathaniel Wesley Filardo
Johns Hopkins University
{jason,nwf}@cs.jhu.edu

COMPLEXITY IN ML SYSTEMS

The complexity of modern ML systems interferes with research, development, and education. It is a truism that an experiment that is casually suggested by a research advisor, and seems to be straightforward, may cost six months before an efficient and (hopefully) bug-free implementation is actually running.

Textbook algorithms may appear relatively simple because they can be written at an abstract level — e.g., as update rules on a small set of nicely notated mathematical quantities. However, applying such an algorithm to a real-world problem means instantiating those abstract quantities in terms of problem-specific data structures that must be efficiently and correctly manipulated.

The abstract quantities mentioned in the textbook might be feature vectors, gradients, random samples, messages, probability distributions (which play many roles including models, posterior distributions, proposal distributions, variational approximations, and stochastic policies), likelihoods, entropies, upper or lower bounds, particles, importance weights, neural activations, weight tensors, loss or reward estimates, etc.

Worse, a typical applied ML system combines multiple modeling/inference/training techniques, so that many types of quantities are interacting. Not only does this increase complexity, but it creates a pressure to optimize across the abstraction boundaries in order to maintain speed. Choosing among possible optimizations is challenging and time-consuming, involving questions such as multiple-use data structures, time-space tradeoffs, loop orders, and use of specialized libraries and hardware. Implementing these optimizations further increases the complexity and risks bugs, particularly as the system evolves during research and development.

THE DYNA LANGUAGE

To insulate users from these implementation choices, we propose an abstract model of programming based on circuits. In our generalized definition, a circuit is a graph that describes the dependence of data items on other data items, using functions and aggregations. We allow the graph to be cyclic or infinite; it can be discovered lazily as computation proceeds.

While our circuit programming model is Turing-complete, it was specifically developed to support design patterns in the applied machine learning (ML) community.

Our user-level circuit language, Dyna [4], allows a concise high-level specification of the heterogeneous computations that are to be performed by a machine learning method. Two examples are shown in Figures 1 and 2. Additional examples are listed in Table 1.

Dyna programs are related to deductive databases with aggregation [3, 9]. A Dyna programmer simply specifies how data items are derived from other data items, perhaps recursively or cyclically. Since the items have *structured names* (similar to Prolog terms), these specifications can usually be cleanly stated by a few schematic rules, which are very close to the textbook ML equations.

The rule notation is inspired by Datalog and Prolog, but Dyna augments these languages with non-boolean values, functional evaluation, and aggregation. For safety, we are currently adding novel mechanisms for type safety, assertion, error handling, and stability of stochastic or nondeterministic computation.

Finally, Dyna supports programming-in-the-large through encapsulation and inheritance. The encapsulation mechanism defines sub-circuit objects called "dynabases" (dynamic deductive databases) that have a public interface. Inheritance is an "extension" mechanism that defines new dynabases as modifications of old ones, augmenting them with new inputs or new rules.

PURE COMPUTATION

Dyna is a pure language, with no side effects and no I/O. A program simply defines a **generalized computational circuit**: a finite or infinite collection of interrelated data items. Rules of the program define each item as the aggregation of some function over other items, which are identified by a pattern-matching syntax. As a base case, some items are simply defined as constants, so a circuit provides a unified interface to *stored data* (these "input" items) along with algorithms for producing *derived data*.

Without I/O, how does one make use of the circuit? An external **driver program** — in a procedural language — uses an API to **update** and **query** the circuit. In other words, the circuit acts like a database or other data structure that is capable of storing information (including derived information) and answering queries about the currently stored information.

Thus, to compile a Dyna program, our proposed system must in effect *synthesize a data structure* (possibly a distributed one) with efficient query and update methods that faithfully support the semantics of the specified circuit.

- Queries may be computationally intensive, since they typically spawn recursive queries via rules. In general, queries return information about a fixpoint of the circuit. A fixpoint is a map from item names to item values, such that each item's value is consistent with its parents' values.[1]
- Updates change the values of input items, affecting the fixpoint. They may be equally computationally intensive if some derived items were memoized to speed up future queries. Updates to input items may render memos stale at derived descendants, so must propagate eagerly or lazily to refresh or flush those memos before they are incorrectly returned.

A single query or update may target all nodes that match a Prolog-style pattern.

[1] If the circuit has multiple fixpoints, then the system is free to use any of them, but a group of queries must answer with respect to the same fixpoint.

2-3 lines	Dijkstra's shortest-path algorithm
4 lines	feed-forward neural network
11 lines	Bigram language model with Good-Turing backoff smoothing
6 lines	Arc-consistency constraint propagation
+6 lines	With backtracking search
+6 lines	With branch-and-bound
6 lines	Loopy belief propagation
3 lines	Probabilistic context-free parsing
+3 lines	Earley's algorithm
+7 lines	Conditional log-linear model of grammar weights (toy example)
+10 lines	Coarse-to-fine A* parsing
4 lines	Value computation in a Markov Decision Process
5 lines	Weighted edit distance
3 lines	Markov chain Monte Carlo (toy example)

Table 1: A range of example ML/AI algorithms whose very short Dyna code is given in [4].

```
phrase(I,K,X) += word(I,K,W) * rewrite(X,W).
phrase(I,K,X) += phrase(I,J,Y) * phrase(J,K,Z)
                 * rewrite(X,Y,Z).
total = phrase(0,sentence_length,"s").
```

Figure 1: A Dyna program specifying a dynamic programming circuit for probabilistic context-free parsing. It defines the `total` item to be the total probability of all parses of a natural-language sentence given as `word` items, under a grammar given as `rewrite` items (which may be either stored or themselves derived from underlying parameters). These 3 rules give the abstract structure of the classical CKY parsing algorithm [11] (or more precisely, the "inside algorithm" [2]), though without specifying a control flow. They respectively say "a word can be a phrase," "two adjacent phrases can form a phrase," and "we seek phrases covering the sentence." They *define* the probability of each hypothesized phrase: `phrase(I,K,X)` denotes the total probability of all sub-parses that span the input substring from `I` to `K`. The second rule can be paraphrased as follows: "If there are possible phrases that span the substrings `(I,J]` and `(J,K]`, of types `Y` and `Z` respectively, and the context-free grammar contains the rule $X \rightarrow Y\ Z$, then deduce that there is a possible phrase of type `X` that spans `(I,K]`. Furthermore, the 'inside probability' of this phrase involves a three-way product." When there are multiple ways to build `phrase(I,K,X)`, its inside probability is defined to sum over all possibilities (choices of `J,Y,Z`), as denoted by the += aggregation operator. To support unary grammar rules $X \rightarrow Y$, we would add the rule `phrase(I,K,X) += phrase(I,K,Y) * rewrite(X,Y)`; this may introduce cycles into the circuit, which must be solved to fixpoint.

EXECUTION

Dyna aims to insulate programmers from the underlying questions of how to store data and schedule computations. We believe that it is a natural abstraction layer for pure computation settings such as ML. Thus, supporting this layer provides a timely challenge to the systems community.

```
sigmoid(X) = 1/(1+exp(-X)).
out(J) = sigmoid(in(J)).
in(J) += out(I) * edge(I,J).
loss += (out(J) - target(J))**2.
```

Figure 2: A feed-forward neural network. Note that line 1 defines infinitely many sigmoid values (computed on demand), line 3 defines a vector-matrix product, and the `loss` training objective in line 4 sums only over output nodes `J` — those for which an item `target(J)` has been defined. The network's structure is specified by defining items of the form `edge(I,J)`, whose values are the edge weights. This could be done by listing one explicit rule per edge. Or it can be done *systematically* by writing edge-defining *rules* in terms of structured node names, where these names will instantiate `I`, `J` above. For example, `edge(input(X,Y),hidden(X+DX,Y+DY)) = weight(DX,DY)` defines a convolutional layer: the node named `hidden(10,10)` receives a connection from the node named `input(8,11)` with weight `weight(-2,1)`.

TensorFlow [1] and PyTorch [8] are also circuit programming models of computation. They cover a subset of Dyna since they are limited to finite, acyclic, feed-forward computation and do not support updates. They also expect users to describe circuits using library operations on a few large data items such as matrices. In contrast, the style of a Dyna program is often more fine-grained — the program might refer directly to individual scalars by structured names, and leave it up to the system to arrange these scalars appropriately into dense matrices and invoke vectorized operations.

Optimizing the execution of TensorFlow and PyTorch programs has been an important accelerant to ML research. The efficient execution of Dyna programs is a generalization of that systems challenge. It is also a generalization of the database systems challenge: since Dyna subsumes Datalog, an implementation of Dyna must make all the same choices as a database system, including storage, indexing, and query cost estimation and planning.

Overall, the space of possible execution strategies for Dyna programs is large and complex [6, 7], involving many free choices for how to schedule work, when to memoize partial results, and what data structures to use for storing memos and input items. The optimal choice of strategy may depend on the input data, the workload of queries and updates, and hardware availability (multiple cores, GPUs, distributed clusters, and memory/disk).

In a recent workshop paper [10], we and our co-authors laid out our plans for gradually tuning the implementation at runtime using reinforcement learning — that is, automatically testing out different mixtures and combinations of strategies under real conditions (using JIT compilation). However, other approaches are possible. For example, static analysis [6] may complement the dynamic analysis performed by reinforcement learning. Our own initial implementation was a Dyna-to-C++ compiler [5] that was limited to efficient forward-chaining execution of semiring-weighted circuits. This was used for a series of 17 NLP papers. We later built a somewhat more general Dyna interpreter that handled to a wider set of circuits, and used this successfully to teach computational linguistics to a class of non-programmers. We encourage other groups to consider how to practically support the full Dyna language or useful subsets.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015).

[2] J. K. Baker. 1979. Trainable Grammars for Speech Recognition. In *Speech Communication Papers Presented at the 97th Meeting of the Acoustical Society of America*, Jared J. Wolf and Dennis H. Klatt (Eds.). MIT, Cambridge, MA.

[3] Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1990. *Logic Programming and Databases*. Springer.

[4] Jason Eisner and Nathaniel W. Filardo. 2011. Dyna: Extending Datalog For Modern AI. In *Datalog Reloaded*, Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers (Eds.). Lecture Notes in Computer Science, Vol. 6702. Springer, 181–220.

[5] Jason Eisner, Eric Goldlust, and Noah A. Smith. 2005. Compiling Comp Ling: Weighted Dynamic Programming and the Dyna Language. (October 2005), 281–290 pages.

[6] Nathaniel Wesley Filardo. 2017. *Dyna 2: Towards a General Weighted Logic Language*. Ph.D. Dissertation. Johns Hopkins University.

[7] Nathaniel Wesley Filardo and Jason Eisner. 2012. A Flexible Solver for Finite Arithmetic Circuits. In *International Conference on Logic Programming (Leibniz International Proc. in Informatics)*, Vol. 17. 425–438.

[8] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic Differentiation in PyTorch. In *Proceedings of the NIPS Autodiff Workshop*.

[9] Kotagiri Ramamohanarao. 1994. Special Issue on Prototypes of Deductive Database Systems. *VLDB* 3, 2 (1994).

[10] Tim Vieira, Matthew Francis-Landau, Nathaniel Wesley Filardo, Farzad Khorasani, and Jason Eisner. 2017. Dyna: Toward a Self-Optimizing Declarative Language for Machine Learning Applications. In *Proceedings of the First ACM SIGPLAN Workshop on Machine Learning and Programming Languages (MAPL)*. ACM, Barcelona, 8–17. http://cs.jhu.edu/~jason/papers/#vieira-et-al-2017

[11] D. H. Younger. 1967. Recognition and Parsing of Context-Free Languages in Time n^3. *Information and Control* 10, 2 (1967), 189–208.