Dyna: A *Non*-Probabilistic Programming Language for Probabilistic AI

Jason Eisner
Department of Computer Science
Johns Hopkins University
Baltimore, MD 21218
jason@cs.jhu.edu

1 Introduction

The Dyna programming language [1, 2, and ongoing work] is intended to provide an abstraction layer for building systems in machine learning and AI. Many of its motivating uses involve probabilistic computation. However, a key decision was to make the language itself non-probabilistic.

2 Approach

A Dyna program is essentially just a set of equational schemata (e.g., recurrence relations) that define named quantities from other named quantities. The program may define infinitely many such quantities, and may express unbounded finite aggregations.

Dyna's formalism is *weighted logic programming*. In an ordinary Prolog logic program, the inference rules (Horn clauses) specify when a term can be proved from other terms. Generalizing this, a Dyna program's inference rules (Horn equations) specify how to derive a term's *value* from the *values* of other terms.[1] Just as a Prolog program defines a set of proved terms under a fixpoint semantics, a Dyna program defines a map from proved terms to values.

A surprisingly large number of systems can be written very concisely in this declarative notation. A prototype version [1] that implements only *semiring-weighted inference* has been used as a key tool in over a dozen NLP papers. A more powerful and efficient version is now under development.

3 Example

For example, here is the inside algorithm for probabilistic context-free parsing. Capital letters are free variables. For each X,I,K, the value of phrase(X,I,K) represents the probability that nonterminal X would rewrite recursively to yield the input substring from I to K. The algorithm can run in $O(\text{sentence\_length}^3)$ time, thanks to reuse of these values, i.e., dynamic programming.

```
1.  phrase(X,I,K) += rewrite(X,W)*word(W,I,K).                     % a word forms a phrase, if X → W
2.  phrase(X,I,K) += rewrite(X,Y,Z)*phrase(Y,I,J)*phrase(Z,J,K).   % so do adjacent phrases, if X → Y Z
3.  phrase(epsilon,I,I) += 1.                                      % empty phrases are available everywhere
4.  result = phrase("S",0,sentence_length).                        % probability of the full sentence
```

Line 3 is an extension to handle epsilons (and it defines infinitely many terms). The above code automatically handles rule cycles such as rewrite("NP","NP",epsilon), as well as lattice parsing. Furthermore, [2] shows that it can be automatically transformed to obtain several other parsing algorithms that are asymptotically more efficient. Moreover, the corresponding outside algorithms are obtained for free by automatic differentiation. By changing += to max=, one gets a Viterbi (1-best) parser. To turn this into an A* parser [3], additional rules can be written to define the priority of updates.

[1]Subterms are evaluated in place by default, not quoted, giving Dyna a functional programming flavor.

4 Beyond probabilities

The values in the above example are probabilities. In general, however, Dyna values may be terms of any type. Even if they are real numbers in the range $[0, 1]$, they do not need to be treated as probabilities. The rules for manipulating them are left up to the programmer.

For example, when using Dyna to implement an A* or best-first search, some values will be only *upper bounds* on probabilities, or other numeric priorities. In MRFs or weighted FSAs, many values are *unnormalized potentials*. In loopy or max-product belief propagation, values are only *approximate* (and perhaps unnormalized) probabilities. In recurrent neural networks, values can be *activations* that decay over time.

Other useful *numeric* values in ML include log-probabilities, annealed probabilities, (expected) losses or rewards, event counts, reusable numerators and denominators, intermediate computations for dynamic programming, transfer functions, regularization terms, feature weights, distances, tunable hyperparameters, and partial derivatives. Dyna programs are free to manipulate all of these.

As Dyna also allows *non-numeric* values, it can handle semiring computations, logical reasoning (including default reasoning), and manipulation of structured objects such as lists, trees, and maps.

5 Benefits

So if Dyna does not commit to a particular probabilistic discipline, in what sense is it designed for probabilistic AI? Like any declarative language, Dyna is meant to encapsulate common chores, handling them efficiently and unobtrusively in support of a concise high-level notation.

Dyna's main job is to determine the values of terms from the inference rules and user input. In other words, it handles inference and search, storage, and indexed lookup, as well as I/O. We argue that it is special-purpose implementations of these tasks that consume the bulk of the code in AI/ML systems and toolkits, which typically run to several tens of thousands of lines in a subfield like NLP.

The new Dyna implementation under development attempts to systematize a wide range of useful strategies for these tasks. Strategies for specific types of terms can be selected by manual declaration. Such choices affect runtime but safely preserve program semantics. Eventually, we plan to use ML to automatically select good strategies for a particular workload.

For example, the implementation specifically supports **mixed inference strategies** (forward and backward chaining with selective or temporary memoization, as well as algebraic shortcuts and powerful program transformations); **a range of data structures** that support efficient storage and fast indexed lookup of relevant terms during inference; **prioritized update propagation**, e.g., updating the left-hand-side of an inference rule after the right-hand side changes;[2] **automatic differentiation** for gradient-based training; and recovery and interactive visualization of **derivation forests**.

Dyna's agnosticism about the semantics of its values is intended to make it a useful layer for building and teaching a *variety* of exact or approximate approaches for specific AI tasks. We hope it can also serve as a useful infrastructure for building toolkits and languages such as those under discussion in this workshop, each of which does commit to some particular semantics.

References

[1] Jason Eisner, Eric Goldlust, and Noah A. Smith. Compiling comp ling: Weighted dynamic programming and the Dyna language. In *Proceedings of HLT-EMNLP*, pages 281–290, 2005.

[2] Jason Eisner and John Blatz. Program transformations for optimization of parsing algorithms and other weighted logic programs. In *Proceedings of the 11th Conference on Formal Grammar*, pages 45–85, 2007.

[3] Dan Klein and Christopher D. Manning. A* parsing: Fast exact Viterbi parse selection. In *Proceedings of HLT-NAACL*, 2003.

[4] P. F. Felzenszwalb and D. McAllester. The generalized A* architecture. *Journal of Artificial Intelligence Research*, 29:153–190, 2007.

[2]Update propagation is the basis of forward-chained inference, memo invalidation, message-passing algorithms such as belief propagation, and dynamic algorithms that recompute outputs as the inputs change. The priority of an update is itself a value computed by the Dyna program (cf. the generalized A* algorithm [4]). In a future version we hope to parallelize by distributing the update queue across multiple processors.