Dyna

Dyna is a high-level “circuit programming” language

Declarative semantics: fixpoint of a circuit

- can be infinitely wide, infinitely deep
- allows cycles
- dynamic data-dependent structure
- Plog-like rules ... very concise!
- Innovative typing and inheritance/modularity features

Prolog-like Rules Define a Dynamic Computation Graph

a = b * c. % equation

Reactive: a keeps up to date with b and c

b += x. equivalent to b = x + y. (almost)

c += z(1). c += z(2). c += z(3). c += z("foo"). Variable

a = b(1) * c(1). % elem-wise multiplication

a += b(1) * c(1). % dot product (sparse)

a(I,K) := b(I,J) * c(J,K). % matrix mult (sparse)

Real-World Examples

Shortest path: distance(V) := distance(U) + edge(U,V).

Probabilistic

Context-free parsing

result := phrase("s", 0, sentence_length).

General Neural Network

a(x) := 1 / (1 + exp(-x)).

out(x) := a(ln(x)).

in(j) := out(i) * edge(i,j).

loss := (out(j) - target(j)) ** 2.

% Convolutional layer in the neural network

edge(input(X), hidden(X+DY), Y) := convweight(input(X), hidden(X), Y).

convweight(input(X), hidden(X), Y) := round(* -1, 1) for DX:-1..1, DY:-1..1.

ML Helping PL: The Abstraction Challenge

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 currently 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.

Abstraction is Key: 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.

Efficiency and Portability: Worse, a typical applied ML system combines several of these 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.

Flexible Solver Architecture

Strategy options

Solvers should systematize all the reasonable implementation tricks that programmers might use and make them work together correctly.

- Parallelizing independent computations
- Ordering dependent computations
- Join strategies
  - Forward vs. backward chaining (update-driven vs. query-driven)
  - Dynamically identify unnecessary computation
- Short-circuiting, branch-and-bound (*), watched variables
- Consolidating related work
  - Static or dynamic batching (consolidating similar tasks, including GPU)
  - Inlining depth (consolidating caller-callee)
- Storage
  - Memorization policy; choose low-level data structures
- Hardware
  - Partitioning the problem across heterogeneous devices (GPUs, distributed)

Reinforcement Learning Objective

Train by actor-critic with fast RLDT actor

Running the Policy

Policy probabilities that are tuned over time (typically approaching 0 or 1).

References and Further Reading