Dyna: Toward a Self-Optimizing Declarative Language for Machine Learning Applications

Tim Vieira  Matthew Francis-Landau  Nathaniel Wesley Filardo  Farzad Khorasani†  Jason Eisner

Johns Hopkins University, USA  †Rice University, USA
{timv,mfl,nwf,jason}@cs.jhu.edu  fk11@rice.edu

Abstract

Declarative programming is a paradigm that allows programmers to specify what they want to compute, leaving how to compute it to a solver. Our declarative programming language, Dyna, is designed to compactly specify computations like those that are frequently encountered in machine learning. As a declarative language, Dyna’s solver has a large space of (correct) strategies available to it. We describe a reinforcement learning framework for adaptively choosing among these strategies to maximize efficiency for a given workload. Adaptivity in execution is especially important for software that will run under a variety of workloads, where no fixed policy works well. We hope that reinforcement learning will identify good policies reasonably quickly—offloading the burden of writing efficient code from human programmers.

CCS Concepts • Theory of computation → Constraint and logic programming; Reinforcement learning; • Software and its engineering → Very high level languages; Just-in-time compilers; Data flow languages

Keywords • Declarative programming; Machine learning; Reinforcement learning

1. Introduction

Many algorithms, especially in machine learning, are set up to compute and maintain a collection of values, iteratively updating them until some convergence criterion has been reached. For example, neural networks, message passing, Gibbs sampling, numerical optimization, and branch-and-bound search can all be defined this way. This makes it possible to design a programming language around rules that define various intermediate and final values in terms of one another and the input values.

Computation graph libraries are close to this view in that a programmer can construct such a graph of relationships among values. But those graphs are specified procedurally. By instead basing our language, Dyna, on logic programming (cf. Prolog (Colmerauer and Roussel 1996)), we get expression pattern matching that can synthesize computation graphs on the fly and in a data-dependent way. The programs can even describe infinite and/or cyclic graphs. A Dyna program has no side effects. Rather, it defines a data structure that can answer queries about the values of nodes in the implicit computation graph. The data structure can also support updates to the values by lazily or eagerly propagating changes to descendant nodes (i.e., truth maintenance), which affects future query answers.

A declarative program of this sort is essentially just a set of equations, which must be jointly solved by traversing relevant parts of the computation graph in order to compute and recompute values as needed. There are many admissible solution strategies that vary in their laziness, in their ordering of tests and nested loops, in their use of parallelism (including GPU kernels), and in how they store and index values. Thus, Dyna provides a playground of optimization opportunities. In principle, a Java or ML program for solving the same system of equations could likewise be optimized, by transforming the source code to change its strategy, but this would in effect require recovering the underlying equations first.

At present, we are designing a new solver for Dyna that will actively explore different execution strategies at runtime, using reinforcement learning. Some strategies for a given program will work better on a given workload, and the solver should migrate toward these over time. This is a technically interesting challenge for reinforcement learning, and has the promise of finding strategies that a human implementer would choose only with prior knowledge of the workload, good judgment, and experimentation.

This workshop paper aims to give an overview of the project and our approach. We first give a high-level sketch of (a subset of) the Dyna language, followed by a quick review of reinforcement learning. We then explain the general architecture of the Dyna runtime solver before elaborating on the optimization opportunities that we face and how we plan to address them with reinforcement learning.

1.1 Dyna at a Glance

Dyna (Eisner and Filardo 2011; Eisner et al. 2005) is a high-level declarative language for succinctly specifying computation graphs via rules. To get a sense for the language, consider a classic example: multi-source shortest path in a directed graph.

\[
\text{pathCost}(S, S) \text{ min}= 0.\\ 
\text{pathCost}(S, T) \text{ min}= \text{pathCost}(S, U) + \text{edge}(U, T).\\ 
\text{edge}(\ast a^*, \ast b^*) = 1. \quad \% \text{hard-coded edge costs}\\ 
\text{edge}(\ast b^*, \ast c^*) = 2.\\ 
\ldots
\]

As the example shows, a Dyna program is a collection of rules given in a Prolog-like notation. Each rule is a template whose variables—named by capitalized identifiers—may be instantiated with arbitrary terms. For example, the first two rules above have infinitely many instantiations, which include...
A cyclic program may not have a unique fixed point. If there is more weight \( \text{weight}(-2,1) \) with these names will instantiate \( I \) where these names will instantiate. We are defining \( \text{pathCost}(\text{"a"},\text{"c"}) \) min = \( \text{min} \), which as in C would compute a running minimum (by analogy to \( ++ \), which as in C would compute a running total). In conventional notation, we are defining \( \text{pathCost}(\text{"a"},\text{"c"}) \equiv \text{min} \) \( \text{pathCost}(\text{"a"},U) + \text{cost}(U,\text{"c"}) \), where the minimization is over all \( U \) such that both of the summands have values. Similarly, \( \text{pathCost}(\text{"a"},\text{"c"}) \) is also defined by a minimization over a bag of values, but in this case the bag also includes 0 thanks to the first rule. Since \( \text{min} \) is an associative and commutative operator, the order in which it aggregates the bag’s elements is immaterial. Dyna programs support general mathematical expressions. A neural network and its training objective can be defined via

\[
\sigma(X) = \frac{1}{1+\exp(-X)}. \quad \% \text{define sigmoid function}
\]

\[
\text{out}(J) = \sigma(\text{in}(J)). \quad \% \text{apply sigmoid function}
\]

\[
\text{in}(J) = \text{out}(I) \times \text{edge}(I,J). \quad \% \text{vector-matrix product}
\]

\[
\text{loss} = (\text{out}(J) - \text{target}(J))^2. \quad \% \text{L2 loss}
\]

The structure and weights of the network are specified by defining the values of items of the form \( \text{edge}(I,J) \). This could be done by listing one rule per edge, but it can also be done systematically by writing edge-defining rules in terms of structured node names, where these names will instantiate \( I, J \) above. For example,

\[
\text{edge}(\text{input}(X,Y), \text{hidden}(X+DX,Y+DY)) = \text{weight}(DX,DY).
\]

\[
\text{weight}(DX,DY) := \text{random}(\ast,-1,1) \text{ for } DX:-4..4, DY:-4..4.
\]

defines a convolutional layer with initially random weights\(^2\) and a \( 9 \times 9 \) convolution filter. For example, the node named \( \text{hidden}(10,10) \) has a connection from the node named \( \text{input}(8,11) \) with weight \( \text{weight}(-2,1) \). The input to the network is specified at runtime by updating the values of \( \text{out}(\text{input}(...) \) ); these items otherwise have no value as the input nodes have no incoming edges.

Dyna’s computation graph may be infinite. For example, the Dyna program for Fibonacci numbers has support over all positive integers.

\[
\text{fib}(0) = 0.
\]

\[
\text{fib}(1) = 1.
\]

\[
\text{fib}(N) = \text{fib}(N-1) + \text{fib}(N-2) \text{ for } N > 1.
\]

To support inference in infinite graphs, the solver should lazily explore only as much of the graph as needed to answer a given query, e.g., a user’s query for the value of \( \text{fib}(13) \).

The Dyna solver seeks a fixed point in which all items are consistent, i.e., each item’s value matches its definition from other values. A cyclic program may not have a unique fixed point. If there is more than one, the solver is permitted to choose arbitrarily. In general, the solver may fail to terminate because no fixed point exists or because it is unable to discover one. Sometimes the solver terminates only at numerical convergence, as for the geometric series sum

\[
\frac{a}{1 + a}.
\]

Eisner and Filardo (2011) give a full explanation of Dyna with more complex examples. Two earlier implementations can be found at dyna.org and github.com/out/dyna. Dyna is designed to handle the kinds of iterative updates needed for AI/ML algorithms in which discovering or updating values will affect other related values (Eisner 2008). Such algorithms include message passing in variational inference and graphical models, constraint propagation in backtracking search, theorem proving (including parsing), neural networks including convolutional networks and backpropagation, Markov chain Monte Carlo, stochastic local search, and so on.

### 1.2 Learning How To Execute: Is It Plausible?

Given a Dyna program, our goal is to automatically search a rich and potentially infinite space of correct implementations. Let us suppose that we can generate code for most of the strategies and reusable tricks (section 3) that human programmers commonly employ. Still, we must automatically decide where to apply these strategies. Will an automatic method actually be able to discover implementations whose efficiency rivals that of human-written code?

Imagine that the competing human programmer (writing in C++) has a Dyna program implicitly in mind. The programmer designs a few data structures, to store the values of a few broad categories of Dyna items. These come with methods for querying and maintaining these values according to the Dyna program’s rules. The programmer mentally generates some implementation options for each item category, and chooses among them by guessing their relative cost. Reinforcement learning is a natural fit for automating such choices. It tries out different options and combinations of options in order to compare their actual cost (i.e., execution profiling). However, how can an automatic approach identify the category of items to which a given option should apply?

Here we can exploit an important property of Dyna: items are not anonymous nodes in some computation graph. They have structured names bestowed by a human programmer. These names are an important hint, allowing the reinforcement learner to learn to classify and compare the items of a given program according to simple features of their names. For example, it might try treating \( \text{pathCost}(\ldots) \) items differently from \( \text{edge}(\ldots) \) items. It can also learn to also pay attention to the arguments of an item’s name. For example, nodes at time step \( T \) in a recurrent neural net might have names that match \( \text{node}(T,1) \), whose computation should be prioritized ahead of nodes whose names match \( \text{node}(T+1,1) \). In short, we expect that execution policies and cost estimation functions can often succeed, for a given Dyna program, by conditioning on relatively simple instance features, of the sort that are commonly induced when learning decision trees (Quinlan 1986) or sparse linear functions (Schmidt and Murphy 2010; Schmidt 2010).

Even experienced human programmers favor simple designs and have limited time to construct and compare alternative implementations. We can therefore entertain the hope that our system may search harder and sometimes discover even better implementations. This is particularly true during research or education, where the humans’ focus would ordinarily be on constructing a variety of Dyna programs for different machine learning models or algorithms, rather than agonizing over the lower-level implementation decisions for each program.

### 2. A Markov Decision Process

We model the adaptive execution of the Dyna solver as a sequential decision-making process, specifically a Markov decision pro-

---

\(^2\)Each expression \( \text{random}(\ast,-1,1) \) names a distinct random variate, since the special argument \( * \) is different in each instantiation of the rule. Thus, the last line defines 81 distinct weights.

\(^3\)Similarly, the \( \text{edges} \) in our computation graph are not anonymous but also have structured names. Each edge corresponds to a particular rule together with a particular binding of values to the rule’s variables. Thus, we can generate different code for different rules. We mention that Dyna also has another form of structured naming: encapsulation features that group together related items and rules into a dyna database (Eisner and Filardo 2011).

\(^4\)And as for less experienced ones, we have often seen new researchers in our lab suffer \( \approx 100 \times \) slowdown by missing common optimizations.
The global time step $t$ is incremented at line 5, but $t$ is used only in the comments (to match the main paper), not elsewhere in the code.

1: while true:  
   \texttt{current time t, current state $s_t$}
2: RUN(AGENDAPOP, device)
3: function AGENDAPOP(device)  
   \texttt{do something from the agenda}
4: \texttt{choose some high-priority task $a_t$ that's appropriate to this device}
5: $a_t \sim \pi(\cdot | \text{curstat}, \text{AGENDAPOP}, \text{device})$; $t \leftarrow t + 1$
6: RUN(method, args) where $a = \text{(method, args)}$
7: return ($a$, none)  
   \texttt{an agenda task returns no value}

\textbf{Listing 2} RUN calls a method that uses policy $\pi$ to choose stochastically among finitely or infinitely many strategies that could correctly execute its task. After execution, RUN adjusts $\pi$ to choose better in future. New code must be generated on demand for a strategy that has not been used before; using existing code (as recorded in $s_t$) is faster, so the policy may learn to avoid asking for novel strategies.

1: function RUN(method, args)  
   \texttt{task or subtask at time t}
2: $(s, C) \leftarrow (\text{curstat}, \text{CU/COST}())$  
   \texttt{save $s_t$, $C_t$ in local vars}
3: $\text{method(args)}$ must sample a strategy $a \sim \pi(\cdot | s_t, method, args)$, then increment $t$ and execute the strategy. Execution may invoke RUN recursively on subtasks (whose methods also sample strategies from $\pi$), so it terminates in state $s_{t'}$ for some $t' > t$.
4: $(a, result) \leftarrow \text{method(args)}$  
   \texttt{denotes the chosen action $a_2$}
5: $\delta C' \leftarrow (\text{curstat}, \text{CU/COST}() - C')$  
   \texttt{new state $s_{t'}$}
6: $\rho(a, result) \leftarrow s_{t'}$ caused $s_t \rightarrow s_{t'}$ costing $C_{t'} - C_t$
7: return result

\textbf{Listing 3} One of the many strategies $\alpha$ that could be chosen and executed during Listing 2, line 4, to carry out a query task on the convolutional neural net of section 1.1. This strategy handles a QUERY-ONEVALUE task with arguments $x'$, $y'$ that seeks the value of the single item in(hidden($x'$, $y'$)). The strategy shown here always recomputes the incoming edges of node hidden($x'$, $y'$). It loops over weight($dx$, $dy$) items having values and similarly invert addition to find the corresponding weight($dx$, $dy$) items having values. Some other example strategies: \(1\) The opposite loop order would loop over out(hidden($x$, $y$)) items having values and similarly invert addition to find the corresponding weight($dx$, $dy$) items having values. However, this strategy is less efficient given our small ($9 \times 9$) convolution filter, since few of the weight($dx$, $dy$) items that we considered would turn out to have values (just those with $-4 \leq dx \leq 4$, $-4 \leq dy \leq 4$). \(2\) If we were not able to invert addition, we would need an even less efficient "guess-and-check" strategy involving nested loops over both kinds of items. \(3\) Before computing the incoming edges, we could check to see if this query has been previously computed and memoized.

\begin{equation}
\rho(\pi) \equiv \mathbb{E} \sum_{t=1}^{\infty} \gamma^t c_t \tag{1}
\end{equation}

where the discount factor $0 < \gamma < 1$ encourages a convergent sum and controls the extent to which the agent cares about the future. Problems with this general form are known as reinforcement learning problems (Sutton and Barto 2017).

In our setting, the MDP is used to control the solution loop shown as Listing 1. The solver maintains an agenda—imagine a priority queue—of tasks that may legally be performed in any order. These tasks are queries and updates of individual items (Filardo and Eisner 2012) or sets of items (e.g., Filardo and Eisner 2017a, b). On each iteration, the solver uses policy $\pi$ to select a task from the agenda, and again uses policy $\pi$ to choose some appropriate strategy to carry it out. Executing this strategy may spawn new queries and updates,\(^5\) to be either executed immediately, or pushed onto the agenda to be executed later. In either case they will be executed by RUN so that $\pi$ can choose a strategy for them, too. During reinforcement learning, $\pi$ evolves over time: it is usually stochastic, and its choice at time $t$ is conditioned on the full current state $s_t$ of the solver. Listings 3–4 show sample strategies for query and update tasks, respectively.

\(6\) This would not be the case if we discounted the cost of action $a_2$ in the traditional way, by $\gamma'$ or $\gamma' \cdot \text{clock}(s_t)$ rather than $\gamma$. In that case, when facing

\(5\) See section 3.1. While some queries and updates come from an external user (see below), the internally generated ones are handled no differently.
features of the state-action pair, which characterize the salient attributes of the decision.

For example, we can use a decision tree (section 4) or an exponential family model,

$$\pi(a \mid s) \propto \exp \left( \theta^T f(s, a) \right),$$

where $f(s, a)$ is a vector of features of the state-action pair, and $\theta$ is a weight vector. $\theta$ can be learned by stochastic gradient descent on a regularized version of $p(\pi)$, as we explain in section 5. As section 1.2 noted, the choice of strategy to handle a query or update task might depend heavily on fast, superficial features of the names of the items being queried and updated. However, a good policy might also consult other features of $s$, e.g., to estimate the cost of executing the strategy $a$ given current data structures, or the impact of executing $a$ on the costs of likely future tasks.

The difficulties of reinforcement learning are primarily:

- **Exploration-exploitation tradeoff:** Our system is not told which action is best in state $s_t$ (in contrast to supervised learning or imitation learning). It must try strategies often enough to find out whether and when they work well. This means spending some time on apparently dubious or trailing strategies.

- **Credit assignment problem:** Determining which actions are responsible for delayed costs or rewards is difficult and may require lengthy experimentation. Listing 2 measures the execution cost $C_{t'} - C_t$ of choosing a strategy $a_t$, but we are also concerned with the impact on the future cost $C_{t'} - C_t$. A strategy might achieve small execution cost only by deferring much work, making it responsible for large future costs when that work is popped from the agenda. A strategy that memoizes the results of its computation may achieve future cost savings when those memos are used to save computation; at the same time, it will also cost something to correct those memos if updates make them stale. A good policy needs to consider these delayed impacts when making a choice, so our reinforcement learner should try to identify the actual impacts of its past choices and distinguish them from confounds and noise.

- **The road not taken:** Credit assignment is ordinarily hard because a reinforcement learner can only try one action $a_t$ at state $t$: it is unable to do a controlled experiment that compares $a_t$ with some other $a_t'$ to get a *paired sample*. In our purely computational setting, however, it would be possible in principle to perform a controlled experiment, by forking the computation or by rolling back to a checkpoint. More practically, because we know the causal structure of our system, there are instances where we can cheaply estimate how some cost $c_{t+k}$ would have been different if we had taken $a_t$ instead of $a_t'$. For example, if we decline to save a computed value and later spend 20ms recomputing it because it was unavailable (Megiddo and Modha 2003), then we know that we could have saved up to 20ms by memoization (less if the memo would have required maintenance or would have been flushed before it was needed).

### 3. Flexible Solution Strategies for Dyna

Conceptually, the Dyna solver operates over a computation graph defined by the Dyna program. Each node is an item named by a Prolog-style ground term (a term with no variables), and the program’s rules define hyperedges that connect these nodes. A task is a request to query or to update a set of nodes. This set can usually be specified by a Prolog-style term, such as `edge(*+::, T)` (which contains a variable $T$).

The solver is free to *memoize* (i.e., store) the value of any derived item in the graph (speeding up future queries of that item and its *descendants*). However, as long as this memo exists, the solver must

---

Listing 4

```
1: w(dx, dy) ← ∆w  \triangleright update the stored (memoized) weight
2: \triangleright now follow each input node's updated outgoing edges to a hidden node, and additively update the latter's \( \sum_s \) value
3: for (out \( \rightarrow \) in) in RUN(QUERY, out(I)):
4: \( (x', y') \leftarrow (x + dx, y + dy) \)
5: AGENDAPUSH(UPDATE, "in(hidden(x', y')) ← o • ∆w")
```

Note that the solver has some interesting flexibility in how it processes the stream. For instance, consider two extremes for update handling. ∆ The solver can eagerly process all updates so that \( s_t \) includes the current contents of the agenda and other data structures. To aid learning, Listing 2 measures how long strategies take to execute. At time step $t$, the policy chooses a strategy $a_t \sim \pi(\cdot | s_t)$ to carry out some task. This *choice* takes time $c_t$ and advances the time counter to $t + 1$, but the process of actually executing strategy $a_t$ may call subroutines that again consult the policy, so that the strategy itself does not complete until some time $t' > t$. At that point, we can measure the total cost $C_{t'} - C_t$ of executing strategy $a_t$, and estimate the value of the new state $s_{t'}$, allowing us to evaluate whether $a_t$ was a good choice and thus update the policy. See section 5 for details.

The MDP’s state space is astronomically large, since $s_t$ includes the current contents of the agenda and other data structures. To help the policy $\pi$ behave reasonably in states that the solver is encountering for the first time, we will define it to depend on a sequence of equal-urgency queries, the solver would have no incentive to answer the current query faster merely in order to start working on the next equally costly query—other than the benefit of finishing the entire stream sooner, which would be negligible since the distant future is heavily discounted. This is why we redefined (1) as (2). If we had started with the “average-reward” variant of (1) (an elegant alternative metric that does not use discounting), we would have correspondingly redefined it in (2) to use the average cost per query, rather than per action.

Note that the solver has some interesting flexibility in how it processes the stream. For instance, consider two extremes for update handling. ∆ The solver can eagerly process all updates so that \( s_t \) includes the current contents of the agenda and other data structures. To aid learning, Listing 2 measures how long strategies take to execute. At time step $t$, the policy chooses a strategy $a_t \sim \pi(\cdot | s_t)$ to carry out some task. This *choice* takes time $c_t$ and advances the time counter to $t + 1$, but the process of actually executing strategy $a_t$ may call subroutines that again consult the policy, so that the strategy itself does not complete until some time $t' > t$. At that point, we can measure the total cost $C_{t'} - C_t$ of executing strategy $a_t$, and estimate the value of the new state $s_{t'}$, allowing us to evaluate whether $a_t$ was a good choice and thus update the policy. See section 5 for details.

The MDP’s state space is astronomically large, since $s_t$ includes the current contents of the agenda and other data structures. To help the policy $\pi$ behave reasonably in states that the solver is encountering for the first time, we will define it to depend on a sequence of equal-urgency queries, the solver would have no incentive to answer the current query faster merely in order to start working on the next equally costly query—other than the benefit of finishing the entire stream sooner, which would be negligible since the distant future is heavily discounted. This is why we redefined (1) as (2). If we had started with the “average-reward” variant of (1) (an elegant alternative metric that does not use discounting), we would have correspondingly redefined it in (2) to use the average cost per query, rather than per action. 

Our code in Listing 4 is somewhat simplified. It should only be called if no `edge` items currently have memos, since the code as shown does not check for such memos and update them.
keep it up to date (slowing down future updates of its ancestors). More generally, the solver can memoize the answer to any query. Such stored-and-maintained query answers are better known as database indexes. For example, the answer to edge(u, v, T) is a traditional adjacency list of outgoing edges, which is useful for computing shortest paths in the opening example of section 1.1.

The framework in the previous section allows the solver to choose among strategies for each query or update task. A mode is a class of queries or updates—an interface—for which we may eventually generate a method that may dispatch to various implementations. For example, one query mode allows any query of the form edge(u, T) where u can be any ground string. To carry out such a query, one would call the corresponding method with u as an argument. A method always begins by consulting the policy to choose a strategy appropriate to its arguments (or sometimes just the parameters of a strategy); it then executes the strategy. A strategy is a piece of code to accomplish the query or update task for the method; it may call other queries and updates as subtasks. The danger is that we may try to call a subtask that does not fall into any mode for which we have a method—which would be a runtime error. We thus check at compile time that we will not get runtime errors for the modes that the Dyna programmer has requested (declared). To be precise, a mode is supported if the solver possesses a complete method for it. A complete method is one that already knows how to handle all necessary subtasks: it never attempts to call a subtask unless that, too, falls into a supported mode. For any query or update mode that was explicitly declared by the Dyna programmer, we are expected to ensure support by constructing a complete method at compile time, which means identifying at least one strategy whose subtask calls are guaranteed (by static analysis) to fall into supported modes. The method may stochastically try to extend itself with additional strategies when it is called at runtime—but as it is supposed to be a complete method, any added strategies must also be complete (or have fallbacks).

We aim to discover good strategies for the modes that arise frequently. All strategies will be essentially specializations of two flexible but incomplete generic methods, which use pattern-matching (unification) against the rules of the program. The generic QUERY method will attempt to implement the most general query mode—handling any query at all—and resembles Prolog’s backward-chaining strategy, SLD resolution (Kowalski 1974). The generic UPDATE method will attempt to implement the most general update mode—handling any update at all—by using forward-chaining to refresh stale memos. This resembles semi-naive bottom-up evaluation akin to Datalog (Ullman 1988; Eisner et al. 2005). See Filardo and Eisner (2012, 2017a,b) for more discussion.

A generic method unifies the query or update against patterns in the Dyna program rules, makes some nondeterministic choices, and recurses to the resulting subtasks. The subtasks are constructed by the unification and may be handled by calls to the generic methods. A strategy is obtained by specializing this generic method to queries or updates of a particular mode, the rules of a particular Dyna program, and a particular branch for each nondeterministic choice. These three specializations narrow the modes that are needed for the subtask calls. This results in a possibly complete strategy that is less flexible but more tightly optimized for its mode. To generalize the method beyond this single strategy, we may restore some of the nondeterminism: we generate code at the start of the method that makes a probabilistic choice a among two or more strategies, based on features of the current state s and their weights \( \theta \). This code defines the policy distribution \( p(\pi | s, \text{method}, \text{args}) \).

If a user interactively attempts a novel kind of query or update at runtime, we can try to generate a supporting strategy on demand. Even when no complete strategy can be found, it is possible to proceed hopefully—as a Prolog interpreter would—by calling the generic method, invoking subtasks without knowing yet whether they will be supportable. However, this may fail (a runtime error).

Even a complete strategy may fail to terminate on an infinite or cyclic computation graph: it knows how to execute its subtasks but it could generate infinitely many of them (either depth-first via recursion or breadth-first via the agenda). Ideally, the solver should detect that it is not making forward progress, and trigger execution of an alternative strategy.

We now discuss some of the options that are available to the generic methods and thus also to their specializations.

3.1 Memoization and Forward/Backward Chaining

In many programs, such as Fibonacci in section 1.1, memoization of some values is extremely important, reducing the runtime from exponential to polynomial thanks to dynamic programming. However, as section 3 noted, memos also have costs—both spatial (memory use) and temporal (creation, access, and maintenance). It is not beneficial to memoize transient values that will never be queried again or that can be quickly recomputed, such as \( \sigma(0.9453) \) in the neural network example of section 1.1. Sometimes memoizing a small fraction of items can be asymptotically effective in both time and space (Zweig and Padmanabhan 2000; Gruslys et al. 2016). A Dyna query method may choose to memoize (or continue memoizing) the answer that it has just obtained, depending on the query strategy chosen by \( \pi \). Other methods could consult \( \pi \) to decide how and when to flush an old memo, e.g., in lieu of updating it, or during garbage collection or adaptive replacement.

Filardo and Eisner (2012) showed that a memoization policy naturally gives rise to a continuum of hybrid execution strategies. A query-driven or lazy solver collects updates to input items but computes nothing until a query arrives, at which point it must work backwards to the input items via more queries. An update-driven or eager solver starts with the updates to input items and immediately finds what it can compute from them, working forward as far as it can via more updates. These traditional backward and forward chaining strategies, respectively, can be viewed as recursively querying unmemorized values and recursively updating memoized values, using the same Dyna rules. Deciding to memoize only some items gives rise to a mix of lazy and eager behavior. An item’s chaining behavior depends on the memoization policy’s past decisions and is not fixed in time.

\( \pi \) should decide not only whether to memoize a result, but in how much detail and using what data structure. Choosing an compact but useful data structure for future use is especially important for large memos such as an index, which might be stored as a sorted list, a hash table, or a neural network.

Footnotes:

8 We ordinarily call the method via RUN (Listing 2) so that the policy will be updated based on what the strategy did and how long it took.

9 Akin to Prolog’s “instantiation fault.” The problem is that built-in items do not support all query modes. E.g., the trick of “running addition backward” in Listing 3 requires us to query for values of \( X \) such that \( X + dx = x’ \). Luckily, in that example, addition is invertible and the built-in implementation knows that the inverse is subtraction. However, operators like \( \text{max} \) are not uniquely invertible and may not support such queries. Also unsupported are queries (and some updates) of an input item or random-number item whose value has been discarded because it was no longer needed to support the declared modes.

10 For simplicity of presentation, the strategies in Listings 3–4 made use of generic methods like QUERY to handle their subtasks, but in practice they would use complete methods specialized to those subtasks.

11 For example, rather than attempting to unroll an infinite left-recursive loop, recursion can try a different subgoal ordering or guess a value and use the agenda to schedule a future revisiting of this guess. Such a transition in strategy does not guarantee convergence (the cycle might, for example, attempt to count to infinity), but it increases the fairness (both disjunctive and conjunctive) of the system by allowing off-cycle items to weigh in.
dense matrix, etc., with differing spatial and temporal consequences (section 3.5). Even when memoizing a simple scalar value, it is sometimes efficient to store the aggregands (e.g., summands) that contributed to that value. This allows faster updates to the memo when the set of aggregands changes slightly, e.g., using a Fenwick tree (Fenwick 1994). Sometimes overhead can be reduced by storing and maintaining only a few of the aggregands, enough to “justify” the current value: so long as these so-called “watched” aggregands do not change, the item’s value is immune to changes in other aggregands (e.g., Moskewicz et al. 2001).

3.2 Prioritization

Listing 1 must choose which task to process next from the agenda. The policy π can choose directly among available tasks, or it can simply pop a high-priority task from a priority queue (or a set of prioritized “bins”), where π was used earlier to choose the priorities.

It is legal to process tasks in any order, but some orders will do less work and answer urgent queries faster. The question of how best to prioritize forward-chaining updates, in particular, turns out to be quite subtle and problem-dependent, which is why we would like to use machine learning to seek a good problem-specific policy.

One principle is that graph structure is important: popping the nodes in a topologically sorted order ensures that each node will only be updated once (barring further external updates). However, the solver may not know a topologically sorted order, since the graph may be large, with a data-dependent structure that is discovered only as the solver runs. Moreover, the graph could be cyclic, in which case no fully topologically sorted order exists.

A competing principle is that updates that will not (strongly) affect the results of open queries should be delayed or omitted. This has motivated A* search heuristics, magic set transformations, and heuristics like residual belief propagation (Elidan et al. 2006).

These two principles may be summarized as (1) “pay attention to items’ names” (which are presumably correlated with the graph structure) and (2) “pay attention to the state of the solver” (including the values of items, the size of updates, and the activity of open queries). In principle, the policy π can learn how to pay attention to both. Specifically, π can use (4) to convert a linear scoring function on tasks to a probability distribution P(π | s) over the agenda tasks.

3.3 Subgoal Order and Rule Order

Dyna is a pure language, like Datalog (Ceri et al. 1989), in that subgoals—the query patterns on the right-hand-side of a rule—have no side-effects. This allows us to visit and instantiate rules in arbitrary order, and to use any strategy for joining a rule’s subgoals, without affecting program semantics.

Rule ordering exposes traditional “short-circuit evaluation” opportunities: An item aggregated by logical OR, having discovered a true aggregand, may cease looking for additional aggregands. This generalizes to other absorbing elements of other aggregators. If an item is likely to have absorbing aggregands, the policy should learn where to look for those, in hopes of finding them quickly. 12

Within the right-hand side of a rule, selecting an order for the subgoal queries corresponds to choosing a loop nesting order, and may have large impact on runtime performance. As a simple example, consider the inner product rule a =+ b(x) * c(x). If b is a large, dense weight vector, while c is a sparse vector of features, then taking c as the outer loop is far more efficient: probing for a few points in b beats probing for (and usually missing) a lot of points in c. Dunlop et al. (2010) shows a more surprising example where a non-standard three-way join order turns out to give a large speedup in context-free natural language parsing. To choose wisely among subgoal orders (consistent with mode constraints), or to estimate their relative cost as a guide to learning, the policy will need to find features that correlate with the runtime, perhaps including actual cardinality estimates from hyperloglog (Flajolet et al. 2007).

3.4 Inlining Depth

The most obvious quantum of work for the solver would be the propagation of a query or an update through a single Dyna rule. While scheduling those quanta would preserve flexibility at runtime, we can reduce overhead by grouping them into larger tasks.

When a strategy needs to call another method, the code generator has three options, which give rise to different variants of the strategy. (As usual, the policy can learn to choose among these strategies.)

1. Generate an AGENDAPUSH instruction to enqueue the method call as a new task on the agenda (where it can be prioritized with respect to other tasks, consolidated with other similar tasks, or picked up by a different compute device). 2. Generate a RUN instruction to immediately call the method as a subtask (which bypasses the overhead of the agenda). 3. Copy the code from one of the method’s strategies (which bypasses the overhead of calling π to choose a strategy at the start of the method). That inlines the subtask into the present strategy, which allows local optimizations across the task-subtask boundary.

Option 3 is at work in the strategies Listings 3–4, each of which propagates through two rules (relating in to edge to weight). These listings also use approaches 1 and 2 at other subtask boundaries.

3.5 Task Structure

The Dyna programmer can, of course, write a program in terms of large objects such as matrices. A single matrix multiplication is a large and coherent task.

However, what if the Dyna programmer chooses to describe the same computation with many small scalar operations? Then another option for the solver is to form larger and more efficient task units. In particular, consider a group of related tasks—such as incrementing the weights of all existing items of the form edge(*b*, *u*), either by the same value or by different values depending on *u*. It is typically more time-efficient to handle these tasks all at once (especially when using parallel hardware such as GPUs or vector units of CPUs). Storing them jointly is also more space-efficient because we do not have to store the repeated structure many times.

To discover related tasks and group them into a single vectorized task, there are three possibilities: 1. Some tasks may be “born” this way, because a strategy runs a loop that generates many related tasks, and the strategy chooses to package them up as a vectorized task. By indexing or partitioning the agenda, we can consolidate related tasks as they are pushed. 2. When a device pops tasks from the agenda (Listing 1), it can scan the agenda for related tasks to run in parallel.

Answers to subgoal queries may also be vectorized. In general, vectorized objects can be represented in multiple ways: (repeated) keys and aggregands, ground keys and associated aggregated value, disjoint non-ground keys, defaults (as per Filardo and Eisner (2017b)), sorted collections, etc. Specialized structures allow for specialized strategies, such as Baeza-Yates (2004)’s algorithm for intersection of two sorted sets.

3.6 Source-to-Source Transforms

Dyna programs are amenable to a number of “source-to-source” transformations (Eisner and Blatz 2007) which can precede, and may facilitate, code generation. These transforms offer forms of inlining, common sub-expression elimination, and other forms of program refactoring. In some cases they improve asymptotic complexity.
4. Decision Tree Policies and Stable Policies

As section 1.2 noted, a method should be able to choose a strategy by applying a few simple and fast tests, as in a decision tree. Garlapati et al. (2015) show that learning a decision tree for classification can itself be elegantly treated as a reinforcement learning problem. A state of their MDP corresponds to a node in a conventional decision tree. It records the results of all tests that have been performed so far on the current input. Their classification agent must then choose among actions: it can either output a specific class (and stop) or perform a specific new test (and transition to a new state according to the result). If the agent’s policy \( \pi \) made a deterministic choice at each state, then it would act like an already-learned decision tree, with each state corresponding to a leaf or internal node according to whether the agent outputs or tests in that state. However, since the MDP is still learning what to do at each state, it acts like a random mixture of decision trees. Over time, it learns to favor decision trees that get high reward (accurate outputs) at low cost (few tests).\(^{13}\)

Classifying Dyna’s structured names imposes some restrictions on test order. E.g., not all names have a second argument, and those that do may store that argument in different places. So a method that tests the second argument of \( \text{edge}(u,t) \) should only be called by a method that knows the functor to be \( \text{edge} \), from a previous test.

We can embed Garlapati et al.’s classification method—RLDT—into Dyna’s control method, using it to select strategies. Listing 2, line 3, is in fact designed to do this. To RUN one of our methods is to visit one of RLDT’s states. This is why our policy \( \pi(a_1 | s_t, \text{method}, \text{args}) \) conditions on the method. The method’s args (perhaps together with the full remaining state \( s_t \) of the Dyna solver) provide the test results, so they correspond to RLDT’s input example. The method immediately samples the action \( a_t \), which may specify either (a) a code strategy to actually carry out the subtask (corresponding to one of RLDT’s output actions), (b) a random choice of appropriate method to be RUN recursively, or (c) a random choice of test (some “if” or “case” statement) whose result (on args or \( s_t \)) will deterministically select an appropriate method to be RUN recursively. The recursive cases will immediately consult the policy again.

**Stable policies:** There is one important setting where it is necessary to make random policy decisions in a repeatable way. Many different dictionary data structures could be used to store memos: for concreteness, suppose we have just a prefix trie \( A \) and a two-dimensional array \( B \). In keeping with our usual “mixed strategy” approach, the reinforcement learner should gradually shift to using whichever one is more efficient. However, if we choose \( A \) to store the memo for item \( x \), we should not later look \( x \) up in \( B \). We must always make the same decision for \( x \).

We still use a decision tree policy to place \( x \). However, here the policy may not look at the transient state \( s_t \), but only at the name of \( x \). Furthermore, rather than drawing a random number, we treat the hash code of the name \( x \) as if it were a uniform random variate. This ensures stability: if \( \pi(A | x) = 0.3 \), we have a 30\% chance of placing \( x \) in \( A \) (determined by the choice of hash function), and will do so consistently.

Although hash(\( x \)) remains stable, there is still a difficulty that \( \pi(A | x) \) may shift to (say) 0.2 during learning. This means that if \( x \) had hash code 0.25, it should now migrate from \( A \) to \( B \). To avoid this, we can take a snapshot \( \pi’ \leftarrow \pi \) of the A-versus-B policy, and continue to act consistently according to \( \pi’ \) even as \( \pi \) continues to learn (see section 5.3 for details). At some point, when \( \pi \) has diverged substantially enough from \( \pi’ \), we take a new snapshot \( \pi'' \leftarrow \pi’ \).

background thread visits all memos in \( A \) and \( B \) and copies each to the other data structure if \( \pi'' \) says it belongs there, marking the original copy for later deletion. After this migration, the solver switches to acting according to \( \pi'' \), and the background thread is free to delete the marked copies. During the migration, queries and updates still act according to \( \pi’ \), but updates to a memo in one data structure must also update the copy (if any) in the other data structure.

5. RL Algorithms

In this section, we will outline one plausible candidate approach to optimizing the long-term cost (2). Our goals are as follows:

- Support online learning, including in the “non-episodic” case where the learner does not run the program multiple times, but must adapt during a single very long run.
- Learn a simple fast policy \( \pi \) that can make each decision using only a few features relevant to that decision, such as simple tests on the arguments to a method call. (Thus, \( \pi \) should not have to estimate the entire future cost as in Q-learning or SARSA.)
- When updating the policy, however, incorporate explicit estimators of quantities such as the execution cost of a strategy. These estimators should also be fairly fast to train and use.\(^{14}\)
- Update the policy probability of strategy \( a_t \) at time \( t’ \) (Listing 2, line 6), immediately after executing it and observing its actual execution cost—without the bookkeeping of measuring its actual impact on future cost \( C_{\infty} - C_{t’} \).
- Instead, estimate the expected future cost following \( a_t \), using appropriate domain-specific features, such as the number and types of memos and tasks in state \( s_{t’} \).

5.1 Cost Estimation

As a warmup, we first adapt the Expected SARSA algorithm (van Seijen et al. 2009) to our setting. Following standard notation, we use the shorthand \( s = s_t, a = a_t, r = \Delta C_t \equiv C_{t’} - C_{t}, s' = s_{t'}, a' = a_{t'} \). The standard algorithm\(^{13}\) takes \( t’ = t + 1 \), but then \( r \) is the cost \( c_t \) of action \( a_t \) alone. Our formulation from section 2 and Listing 2 instead takes \( t’ \) to be the time upon completing the execution of the strategy selected at \( a_t \), so that \( r \) measures the total cost of selecting and executing the strategy.

\[ Q_x(s,a) \text{ is defined to be the expected long-term (discounted) cost if we start in state } s, \text{ select strategy } a, \text{ and follow } \pi \text{ thereafter.} \]

In the classical formulation (1) of long-term cost, we would define

\[ Q_x(s,a) \equiv \mathbb{E} \left[ \sum_{t=1}^{\infty} \gamma^t c_t \mid s_1 = s, a_1 = a \right] \]

Given our revised formulation (3) of long-term cost, we omit \( \gamma^t \) because discounting is now included within the definition of \( c_t \):\(^{16}\)

\[ Q_x(s,a) \equiv \mathbb{E} \left[ \frac{1}{\gamma(I(s))} \sum_{t=1}^{\infty} c_t \mid s_1 = s, a_1 = a \right] \]

where \( I(s) \) is the number of external queries that arrived prior to state \( s \), some of which may still be contained in the set \( \mathcal{O}(s) \) of open queries (so the next external query will be numbered \( i = I(s) + 1 \)).\(^{16}\)

\(^{13}\)As an extension of this method, we believe, one could learn to include only a high-reward, low-cost, dynamic subset of the features in a linear scoring function (cf. Strubell et al. 2015; He et al. 2016). Such functions are needed for reward estimation \( r(s,a) \) in section 5.4, and for policies based on exponential-family distributions like (4) rather than decision trees.

\(^{14}\)Training will nonetheless slow down execution. The previous bullet point tries to ensure that running the policy is fast once we are no longer training. If we never want to stop training, we can still speed up execution as the policy converges by (intelligently) subsampling a subset of time steps on which to compute parameter updates.

\(^{15}\)The standard formulation also takes \( r \) and \( Q \) to be rewards to be maximized, whereas for us they are costs to be minimized.

\(^{16}\)The need to divide by \( \gamma(I(s)) \) is a subtle point. Otherwise \( Q_x(s,a) \) would be very small for states \( s \) with large \( I(s) \), as the summed costs...
We measure the current error of while executing policy because the path has not yet reached the stationary distribution $\gamma$ in effect dividing by and thus its cost should be weighted by $\gamma$. SARSA’s least-squares estimation of $Q(\pi)$ for a given method. Thus entire future cost, $\pi_s$ features of $Q_\pi$ to move $\hat{r}$ of $Q(s,a)$ is the advantage $Q(s,a) - V(s)$. SARSA's least-squares estimation of $Q(\pi)$ for a given method. Thus entire future cost, $\pi_s$ features of $Q_\pi$ to move $\hat{r}$ of $Q(s,a)$ is the advantage $Q(s,a) - V(s)$. SARSA's least-squares estimation of $Q(\pi)$ for a given method. Thus entire future cost, $\pi_s$ features of $Q_\pi$ to move $\hat{r}$ of $Q(s,a)$ is the advantage $Q(s,a) - V(s)$. SARSA's least-squares estimation of $Q(\pi)$ for a given method. Thus entire future cost, $\pi_s$ features of $Q_\pi$ to move $\hat{r}$ of $Q(s,a)$ is the advantage $Q(s,a) - V(s)$. SARSA's least-squares estimation of $Q(\pi)$ for a given method. Thus entire future cost, $\pi_s$ features of $Q_\pi$ to move $\hat{r}$ of $Q(s,a)$ is the advantage $Q(s,a) - V(s)$. SARSA's least-squares estimation of $Q(\pi)$ for a given method. Thus entire future cost, $\pi_s$ features of $Q_\pi$ to move $\hat{r}$ of $Q(s,a)$ is the advantage $Q(s,a) - V(s)$ from the previous section. We obtain the following approximate stochastic gradient update to $\theta$ at time $t$: while $\eta_t$ is an optimization stepsize. In short, Listing 2, line 6, should first update the parameters of $\hat{Q}$ and $V$ as described in the previous section, and then use equation (11) to update the parameters of $\pi$. Intuitively, the latter update aims to increase $\pi$’s future probability of again choosing $a_t$ in state $s_t$ if and only if $a_t$'s cost is now estimated to be lower than average, i.e., $\hat{Q}(s_t, a_t) < \hat{V}(s_t)$. 5.3 Learning Off-Policy In section 4, we faced a scenario where we had to sample actions according to an older policy $\pi'$, fixing $\pi'$ for a period of time. However, we would like to continue to train $\hat{Q}, \hat{V}$, and $\pi$ by stochastically following the same gradients as if we had been sampling from $\pi$. This can be done by importance reweighting. Basically, when we have followed a path under $\pi'$ that would have been more (or less) likely under $\pi$, we should correspondingly upweight (or downweight) the resulting updates. As we follow the path under $\pi'$, we maintain its cumulative importance weight $\omega_t = \omega_{t-1} \cdot \pi(a_t|s_t)/\pi'(a_t|s_t)$ at all times $t$, where $\omega_0 = 1$. For the updates at time $t$ (Listing 2, line 6), the stochastic gradient formulas in the previous two sections must be corrected by multiplying them by $\omega_t$. 5.4 Estimation by Linear Regression We must train an estimator $\hat{V}(s)$ of expected future cost. Many estimator families are possible, but a simple one would be the linear functions $\langle \text{load}(s) \cdot w \rangle \Phi(s)$, where $w$ is a vector of trainable parameters and $\Phi(s)$ is a vector of numerical features synthesized by the Dyna compiler (including a bias feature). $\Phi$ can be efficiently maintained as the solver runs, using sparse incremental updates. For example, a feature $\Phi_a(s)$ might count how many tasks on the agenda that use method $k$, so $\text{load}(s) \cdot w_k$ should estimate the total discounted future cost of those tasks and their progeny. Or $\Phi_b(s)$ might count how many memos currently exist of a certain kind, in which case $\text{load}(s) \cdot w_k$ would represent something like the future incurred cost minus future saved cost per memo (hopefully $< 0$). Since $c_t$ is proportional to $\text{load}(s_t)$, we have included a factor of $\text{load}(s)$ in our cost estimator, on the theory that a higher load at present predicts a proportionately higher load when the tasks are executed. That is plausible if the tasks that contribute most to $\hat{V}(s)$ will be executed soon (which is likely because of temporal discounting). We can model $\hat{Q}(s,a)$ by explicit lookahead to $\hat{V}(s')$, as suggested by equation (8). That is, we use the linear function $\hat{Q}(s, a) \equiv 1 \cdot \hat{r}(s, a) + \langle \text{load}(s) \cdot w \rangle \Phi(s)$. Provided that these two estimators are unbiased, then (8) automatically holds between the estimates; $\hat{Q}$ has no other parameters to tune. In
practice, observing that \( \Phi(s) = (\frac{\Delta \text{load}(s)}{\text{load}(s)} + 1) \cdot \gamma^t \cdot (\Phi(s) + \Delta \Phi) \), we can define our prediction \( \Phi \) to take the same form, using separate estimators of the feature change vector \( \Delta \Phi \) and the changes to the open queries, \( \Delta I \overset{\text{def}}{=} I(s') - I(s) \) and \( \Delta \text{load} \overset{\text{def}}{=} \text{load}(s') - \text{load}(s) \). Linear regression estimators of \( r \) and these \( \Delta \) values can consult \( a \) and the features \( \Phi(s) \), and can be trained online, using supervised observations that are available at time \( t' \).

We suspect that we can make this design fast even when \( \Phi \) includes many features. The key point is that most actions have limited ability to change the state: hence the actual feature change \( \Delta \Phi \) and our predicted change \( \Delta \Phi(s, a) \) will both be sparse vectors. As a result, computations like \( \hat{A}(s, a) \) (needed by (11)) are similar across nearby timestamps and can share work. It also follows that updates to linear regression weight vectors (including \( w \)—see footnote 18) will themselves tend to be sparse, either in the sense that they are mostly zero or in the sense that they are mostly identical to a scaled version of the previous update. Hence we can use sparse vector updating tricks (Carpenter 2008), including storing explicit scalar multipliers and using timestamping to defer repeated similar updates so that they can be performed as a batch. Indeed, applying such tricks automatically to user programs is one goal of Dyna.

6. Related Work

Adaptive performance tuning is not a new idea. Branch prediction is a specification of a data structure as a set of operations it must support, which have well-defined semantics. In just this way, a Dyna program can specify a set of supported queries and updates and their semantics. The Dyna solver then provides a flexible, self-tuning implementation of this ADT.

There are other examples of self-tuning implementations for more specific ADTs. Many applications find that a custom-written dispatch policy that selects among several implementations can significantly improve performance. In addition, data structures can have control parameters that allow them to pursue mixed strategies (as in our design), interpolating between certain extremes. A great example of self-tuning control parameters is the adaptive replacement cache (ARC) (Megiddo and Modha 2003)—a fixed-size cache data structure with a clever adaptive policy that determines what to evict from the cache to optimize the hit rate (frequency that a requested item is in the cache). This relates to our problem of deciding what to memoize (although we also face the additional problem of maintaining menus). Another example is SmartLocks (Eastep et al. 2010), a reinforcement learning mechanism for self-tuning locking mechanisms in parallel applications.

7. Conclusion

Dyna is a general-purpose language that allows a programmer to synthesize computation graphs. Such a graph defines how to compute and maintain derived data. At an abstract level, this is the focus of all machine learning systems. We intend Dyna as a practical vehicle for concise declarative specification of real-world machine learning computations.

Our previous Dyna implementations (solvers) used homogeneous strategies to compute, store, and maintain data items. However, a solver has considerable flexibility (Filardo and Eisinger 2012) about how to store the derived data and when and how to handle queries and updates. These provide a range of optimization opportunities. In this paper, we have outlined a possible reinforcement learning architecture for exploring a mix of strategies at runtime, shifting probability toward strategies that seem to have lower long-term cost.

Acknowledgments

This material is based upon work supported by the National Science Foundation under collaborative Grants No. 1629564 and 1629459.