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
{timv,mfl,nwf,jason}@cs.jhu.edu

†Rice University, USA
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[1] 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.

```
pathCost(S,S) min= 0.
pathCost(S,T) min= pathCost(S,U) + edge(U,T).

edge("a","b") = 1.                    % hard-coded edge costs
edge("b","c") = 2.
 ...
```

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

[1] E.g., Theano (Theano Development Team 2016) or TensorFlow (Abadi et al. 2015).

MAPL'17, June 18, 2017, Barcelona, Spain
ACM. 978-1-4503-5071-6/17/06...$15.00
http://dx.doi.org/10.1145/3088525.3088562

```
pathCost("a","a") min= 0.
pathCost("a","a") min= pathCost("a","b") + edge("b","a").
pathCost("a","c") min= pathCost("a","b") + edge("b","c").
pathCost("a","c") min= pathCost("a","d") + edge("d","c").
 ...
```

These instantiations collectively define the **values** of many **items** such as `pathCost("a","c")`. The **aggregator** `min=` computes each item using a running minimum (by analogy to `+=`, which as in C would compute a running total). In conventional notation, we are defining `pathCost("a","c")` $\stackrel{\text{def}}{=} \min_{\texttt{U}}$ `pathCost("a",U) + cost(U,"c")`, where the minimization is over all `U` such that both of the summands have values. Similarly, `pathCost("a","a")` 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 `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

```
σ(X) = 1/(1+exp(-X)).              % define sigmoid function
out(J) = σ(in(J)).                  % apply sigmoid function
in(J) += out(I) * edge(I,J).        % vector-matrix product
loss += (out(J) - target(J))**2.                % L2 loss
```

The structure and weights of the network are specified by defining the values of items of the form `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,

```
edge(input(X,Y),hidden(X+DX,Y+DY)) = weight(DX,DY).
weight(DX,DY) := random(*,-1,1) for DX:-4..4, DY:-4..4.
```

defines a convolutional layer with initially random weights[2] and a 9×9 convolution filter. For example, the node named `hidden(10,10)` has a connection from the node named `input(8,11)` with weight `weight(-2,1)`. The input to the network is specified at runtime by updating the values of `out(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.

```
fib(0) = 0.
fib(1) = 1.
fib(N) = fib(N-1) + fib(N-2) 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 `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

```
a += 1.
a += a/2.
```

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/nwf/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.

[2] Each expression `random(*,-1,1)` names a distinct random variate, since the special argument `*` generates a symbol ("gensym") that is different in each instantiation of the rule. Thus, the last line defines 81 distinct weights.

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.[3] 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 `pathCost(...)` items differently from `edge(...)` 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 `node(T,I)`, whose computation should be prioritized ahead of nodes whose names match `node(T+1,I)`. 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.[4] 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-

[3] Similarly, the *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 **dynabase** (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.

Listing 1 Each thread controlling a compute device (CPU core or GPU) runs this loop to request pending tasks from a global agenda, execute them (which may push new tasks back onto the agenda), and determine how long this took. The learned policy π stochastically chooses an agenda task and executes it using RUN (Listing 2). *Note:* 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 :                                   ▷ current time t, current state s_t
2:     RUN(AGENDAPOP, device)
3: function AGENDAPOP(device)                     ▷ do something from the agenda
4:     ▷ choose some high-priority task a_t that's appropriate to this device
5:     a ~ π(· | currstate, AGENDAPOP, device); t += 1
6:     RUN(method, args) where a = (method, args)
7:     return (a, none)                           ▷ an agenda task returns no value
```

Listing 2 RUN calls a method that uses policy π to choose stochastically among finitely or infinitely many strategies that could correctly execute its task. After execution, RUN adjusts π 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)                     ▷ task or subtask at time t
2:     (s, C) ← (currstate, CUCOST())             ▷ save s_t, C_t in local vars
3:     ▷ method(args) must sample a strategy a ← a_t ~ π(· | s_t, method,
         args), then increment t and execute the strategy. Execution may
         invoke RUN recursively on subtasks (whose methods also sample
         strategies from π), so it terminates in state s_t' for some t' > t.
4:     (a, result) ← method(args)                 ▷ a denotes the chosen action a_t
5:     (s', ΔC) ← (currstate, CUCOST() − C)       ▷ new state s_t'
6:     π.learn(s, a, ΔC, s')                      ▷ a_t caused s_t ⇝ s_t' costing C_t' − C_t
7:     return result
8: function CUCOST()                              ▷ get cumulative cost C_t if currstate = s_t
9:     C += load(currstate) · (CLOCK() − tick)    ▷ add c_{t−1} (see (3))
10:    tick ← CLOCK()                             ▷ record wall-clock time of s_t for next call
11:    return C                                   ▷ C and tick are static local variables
```

cess (MDP). As execution encounters **choice points** in the code, where multiple choices (**actions**) are available, a **policy** π determines an appropriate action $a_t \sim \pi(\cdot|s_t)$ given the **state** s_t of the system at time t. Upon executing the action, the solver **transitions** to the next state s_{t+1} and observes a scalar **cost** c_t, $(s_{t+1}, c_t) \sim p(\cdot \mid s_t, a_t)$. One typically seeks a policy that minimizes the ***long-term*** **discounted cost**,

$$\rho(\pi) \stackrel{\text{def}}{=} \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^t c_t\right] \tag{1}$$

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 π to select a task from the agenda, and again uses policy π 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 π can choose a strategy for them, too. During reinforcement learning,

Listing 3 One of the many strategies a 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 QUERYONEVALUE 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 "runs addition backwards" (subtraction) to find the corresponding `out(input(`x`,`y`))` items having values. Some other example strategies: ① The opposite loop order would loop over `out(input(`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 (9×9) 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.

```
1: i ← NULL                                       ▷ Aggregation identity element
2: ▷ compute all summands to hidden(x',y') item
3: for (weight(dx,dy) ↦ w) in RUN(QUERY, weight(DX,DY)) :
4:     (x, y) ← (x' − dx, y' − dy)                ▷ so that x' = x + dx, etc.
5:     if o ← RUN(QUERYONEVALUE, out(input(x,y))) :
6:         i += o · w
7: return (a, i)                                  ▷ must return the chosen strategy a along with answer
```

π 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.

What cost should we minimize? Given a Dyna program, a **driver program** interacts with the resulting data structure by issuing a stream of external queries and updates (which place work onto the agenda—not shown in Listing 1). Queries may be synchronous (blocking) or asynchronous (non-blocking), but the solver must always provide a correct answer to each query with respect to the updates issued before it. For taking time to answer the i^{th} query, the solver incurs a cost of $\lambda_i \cdot \text{latency}(i)$, where λ_i is the urgency of the query (specified by the driver program in units of cost/second) and $\text{latency}(i)$ is the elapsed wall-clock time between the query and its answer (known as "flow time" in the job scheduling literature). We now *redefine* the long-term discounted cost from (1) to be

$$\rho(\pi) \stackrel{\text{def}}{=} \mathbb{E}\left[\sum_{i=1}^{\infty} \gamma^i \lambda_i \cdot \text{latency}(i)\right] \tag{2}$$

where $0 < \gamma < 1$ means that at any time, the solver cares more about minimizing latency of currently open queries—especially the oldest ones—than future queries. Taking $\mathcal{O}(s_t)$ to denote the set of open external queries in state s_t, we define the solver's **load** at time t, in units of (discounted) cost/second, to be $\text{load}(s_t) \stackrel{\text{def}}{=} \sum_{i \in \mathcal{O}(t)} \gamma^i \lambda_i$. We can now rearrange equation (2) to sum over time steps t—each of which extends the latencies of $\mathcal{O}(s_t)$—rather than over queries i:

$$\rho(\pi) = \mathbb{E}\left[\sum_{t=1}^{\infty} c_t\right] = \mathbb{E}\left[\lim_{t \to \infty} C_t\right] \text{ for } C_t \stackrel{\text{def}}{=} c_1 + \cdots + c_{t-1}$$

$$c_t \stackrel{\text{def}}{=} \text{load}(s_t) \cdot (\text{clock}(s_{t+1}) - \text{clock}(s_t)) \tag{3}$$

where $\text{clock}(s_t)$ is the wall-clock time upon entering state s_t. Notice that it is in the solver's interest to answer open queries: closing a query will usually achieve a reduction in load, even if the driver program reacts by issuing its next query (since the new query has larger i and hence smaller γ^i than the query it replaces).[6]

[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.

[6] This would not be the case if we discounted the cost of action a_t in the traditional way, by γ^t or $\gamma^{\text{clock}(s_t)}$ rather than γ^i. In that case, when facing

Listing 4 Just as Listing 3 shows one strategy for an query task, here we see one strategy for an update task—an additive weight update "`weight(`dx`,`dy`) +=` Δw" when training the convolutional neural net of section 1.1. This particular strategy chooses to propagate the additive update through *two* rules, computing the additive effect on various `edge` items[7] and then on the `in` items reached by those edges. This code does not itself modify the `in` items or propagate the change to *their* descendants—instead, that work is handled in the final line as a separate update task (which will get to choose its own strategy when it is later popped from the agenda and RUN). What are some alternative strategies? ① Execute the final line's task immediately via RUN(UPDATE, ...), instead of deferring it via AGENDAPUSH(...). ② Change the final line's update to "`in(hidden(`x'`,`y'`)) ←` `UNKNOWN`", which would invalidate the memoized value of that item rather than incrementing it. Then the memo no longer needs to be maintained, but must later be recomputed from scratch upon lookup. ③ A more aggressive strategy would bulk-invalidate *all* memoized `in(hidden(...))` values, rather than iterating over just those hidden nodes that were provably affected by the weight change as below. This is faster and may turn out to work just as well, since in practice, almost all hidden nodes will indeed be affected. ④ Realize the entire loop "all at once" as a matrix operation, shifting and scaling the `out(input(...))` matrix and adding it to the `in(hidden(...))` matrix. This is possible for portions of these matrices that are currently stored in dense arrays.

```
1: w[dx, dy] += Δw                          ▷ update the stored (memoized) weight
2: ▷ now follow each input node's updated outgoing edge to a hidden
     node, and additively update the latter's in(...) value
3: for (out(input(x,y)) ↦ o) in RUN(QUERY, out(I)):
4:     (x', y') ← (x + dx, y + dy)
5:     AGENDAPUSH(UPDATE, "in(hidden(x',y')) += o·Δw")
```

Note that the solver has some interesting flexibility in how it processes the stream. For instance, consider two extremes for update handling. ① The solver can eagerly process all updates so that subsequent queries have low latency. ② The solver can buffer the updates and lazily apply them only when they are needed to correctly answer some query. This reduces the latency of current queries and may wholly eliminate work that turns out to be unneeded.

To aid learning, Listing 2 measures how long strategies take to execute. At time step t, the policy chooses a strategy $a_t \sim \pi(\cdot \mid s_t)$ 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 π behave reasonably in states that the solver is encountering for the first time, we will define it to depend on **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,

$$\pi(a \mid s) \propto \exp\left(\theta^\top f(s,a)\right), \tag{4}$$

where $f(s,a)$ is a vector of features of the state-action pair, and θ is a weight vector. θ can be learned by stochastic gradient descent on a regularized version of $\rho(\pi)$, 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** $C_\infty - C_{t'}$. 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 $\bar{a}_t$ 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 $\bar{a}_t$ 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("b",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

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*.

[7] 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("b",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.[8] 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.[9] 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.[10] 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 ① queries or updates of a particular mode, ② the rules of a particular Dyna program, and ③ 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 θ. This code *defines* the policy distribution $\pi(a \mid s, \text{method}, \text{args})$.

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.[11]

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 π. Other methods could consult π 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 unmemoized 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.

π 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

[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 $\texttt{X} + dx = x'$. Luckily, in that example, addition is invertible and the built-in implementation knows that the inverse is subtraction. However, operators like $\max$ are not uniquely invertible and may not support such queries. Also unsupportable 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.

① 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.

② 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 $\pi(a \mid s)$ 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).

[12] It may also be possible to find and exploit short-circuiting opportunities among the items within a rule, though the details are more complicated than across rules.

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. ② By indexing or partitioning the agenda, we can consolidate related tasks as they are pushed. ③ 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 π 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 `edge(`u`,`t`)` should only be called by a method that knows the functor to be `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 $\pi(a_t \mid s_t, \text{method}, \text{args})$ 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 ① a code strategy to actually carry out the subtask (corresponding to one of RLDT's output actions), ② a random choice of appropriate method to be RUN recursively, ③ 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 $\pi(\text{A} \mid x) = 0.3$, we have a 30% chance of placing x in A (determined by the choice of hash function), and will do so consistently.

Although $\text{hash}(x)$ remains stable, there is still a difficulty that $\pi(\text{A} \mid x)$ 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 π' even as π continues to learn (see section 5.3 for details). At some point, when π has diverged substantially enough from π', we take a new snapshot $\pi'' \leftarrow \pi$. A background thread visits all memos in A and B and copies each to the other data structure if π'' says it belongs there, marking the original copy for later deletion. After this **migration**, the solver switches to acting according to π'', and the background thread is free to delete the marked copies. During the migration, queries and updates still act according to π', but updates to a memo in one data structure must *also* update the copy (if any) in the other data structure.

[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 $\hat{r}(s, a)$ in section 5.4, and for policies based on exponential-family distributions like (4) rather than decision trees.

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 π 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, π 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 $s = s_t, a = a_t, r = \Delta C \stackrel{\text{def}}{=} C_{t'} - C_t, s' = s_{t'}, a' = a_{t'}$. The standard algorithm[15] 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.

$Q_\pi(s, a)$ is defined to be the expected long-term (discounted) cost if we start in state s, select strategy a, and follow π thereafter. In the classical formulation (1) of long-term cost, we would define

$$Q_\pi(s, a) \stackrel{\text{def}}{=} \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^t c_t \;\middle|\; s_1 = s, a_1 = a\right] \tag{5}$$

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

$$Q_\pi(s, a) \stackrel{\text{def}}{=} \frac{1}{\gamma^{I(s)}} \cdot \mathbb{E}\left[\sum_{t=1}^{\infty} c_t \;\middle|\; s_1 = s, a_1 = a\right] \tag{6}$$

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]

[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 $Q_\pi(s, a)$ would be very small for states s with large $I(s)$, as the summed costs

By writing the expectation within (6) as a recurrence, we obtain

$$Q_\pi(s,a) \;=\; \mathbb{E}\left[r \;+\; \gamma^{I(s')-I(s)} \cdot Q_\pi(s',a')\right] \qquad (7)$$

where we have redefined $r \stackrel{\text{def}}{=} \Delta C/\gamma^{I(s)}$. (Our previous redefinition $r \stackrel{\text{def}}{=} \Delta C$ dealt with our generalization that allowed $t' \neq t+1$, but not with our revised long-term cost.) We further rewrite as

$$Q_\pi(s,a) \;=\; \mathbb{E}\left[r \;+\; \gamma^{I(s')-I(s)} \cdot V_\pi(s')\right] \qquad (8)$$

$$\text{where} \qquad V_\pi(s) \stackrel{\text{def}}{=} \mathbb{E}_{\bar{a}\sim\pi(\cdot|s)}\left[Q_\pi(s,\bar{a})\right] \qquad (9)$$

While executing policy π, we can use temporal difference updating to improve parametric approximations $\widehat{Q}, \widehat{V}$ to the functions Q_π, V_π. We measure the current error of $\widehat{Q}$ by $\mathbb{E}_{s,a,r,s'}[\frac{1}{2}(\widehat{Q}(s,a)-q)^2]$ where $q \stackrel{\text{def}}{=} \left(r + \gamma^{I(s')-I(s)} \cdot \widehat{V}(s')\right)$ is the "target" value. The error derivative with respect to $\widehat{Q}(s,a)$ is $\mathbb{E}_{s,a,r,s'}[\widehat{Q}(s,a)-q]$, and we can get an unbiased estimate by observing the **temporal difference** $\delta \stackrel{\text{def}}{=} \widehat{Q}(s,a)-q$ on any tuple (s,a,r,s'). Thus, tuning the parameters of $\widehat{Q}$ by stochastic gradient descent simply tries to move $\widehat{Q}(s,a)$ closer to q. Similarly, we can tune $\widehat{V}$ using the estimated derivative $\widehat{V}(s)-q$, moving $\widehat{V}(s)$ closer to q.

5.2 Policy Learning

SARSA learning would define the policy $\pi(a \mid s)$ in terms of $\widehat{Q}$, placing high probability on actions a with low estimated cost $\widehat{Q}(s,a)$. Thus, updating $\widehat{Q}$ would automatically update the policy.

We instead propose to use $\widehat{Q}$ as a signal to train the parameters of a separate, decoupled policy π (Sutton et al. 2000; Bhatnagar et al. 2007). Our rationale is that while $\widehat{Q}$ may need to use complex features of s (such as the contents of the agenda) to estimate the entire future cost, π only needs to choose among a few strategies for a given method. Thus π can have a much simpler form in terms of features locally available to the method call, making it fast.

Our stochastic policy π will be smoothly parameterized by θ. The policy gradient theorem (Sutton et al. 2000, Theorem 1) states

$$\nabla_\theta \rho(\theta) = \mathop{\mathbb{E}}_{\substack{s\sim d_\pi(\cdot)\\ a\sim\pi(\cdot|s)}}\left[A_\pi(s,a)\;\nabla_\theta \log \pi(a \mid s)\right] \qquad (10)$$

where $\rho(\theta)$ is the long-term discounted cost (3), d_π is the (stationary) distribution of states visited by the policy π, and $A_\pi(s,a) \stackrel{\text{def}}{=} Q_\pi(s,a) - V_\pi(s)$ is the **advantage** of action a in state s.

In theory, we could estimate (10) by Monte Carlo, sampling one or more states s from d_π. In practice, RL algorithms approximate such sampling by taking s to be consecutive samples along an execution path.[17] In our case, this means running Listing 1 with policy π to see which states s_t it encounters and which actions a_t it takes. Replacing $A_\pi(s,a)$ with the approximation $\widehat{A}(s,a) \stackrel{\text{def}}{=} \widehat{Q}(s,a) - \widehat{V}(s)$ from the previous section, we obtain the following approximate stochastic gradient update to θ at time t:

$$\theta \;\leftarrow\; \theta - \eta_t \cdot \widehat{A}(s_t,a_t)\;\nabla_\theta \log \pi(a_t|s_t) \qquad (11)$$

where η_t is an optimization stepsize. In short, Listing 2, line 6, should first update the parameters of $\widehat{Q}$ and $\widehat{V}$ as described in the previous section, and then use equation (11) to update the parameters θ of π. Intuitively, the latter update aims to increase π'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., $\widehat{Q}(s_t,a_t) < \widehat{V}(s_t)$.

5.3 Learning Off-Policy

In section 4, we faced a scenario where we had to sample actions according to an older policy π', fixing π' for a period of time. However, we would like to continue to train $\widehat{Q}, \widehat{V}$, and π by stochastically following the same gradients as if we had been sampling from π.

This can be done by importance reweighting. Basically, when we have followed a path under π' that would have been more (or less) likely under π, we should correspondingly upweight (or downweight) the resulting updates. As we follow the path under π', we maintain its cumulative importance weight $\omega_t = \omega_{t-1} \cdot \frac{\pi(a_t|s_t)}{\pi'(a_t|s_t)}$ 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 $\widehat{V}(s)$ of expected future cost. Many estimator families are possible, but a simple one would be the linear functions $(\text{load}(s)\cdot\mathbf{w})^\top\mathbf{\Phi}(s)$, where $\mathbf{w}$ is a vector of trainable parameters and $\mathbf{\Phi}(s)$ is a vector of numerical features synthesized by the Dyna compiler (including a bias feature). $\mathbf{\Phi}$ can be efficiently maintained as the solver runs, using sparse incremental updates.

For example, a feature $\Phi_k(s)$ might count the number of 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 $\Phi_k(s)$ 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 $\widehat{V}(s)$ will be executed soon (which is likely because of temporal discounting).

We can model $\widehat{Q}(s,a)$ by explicit lookahead to $\widehat{V}(s')$, as suggested by equation (8). That is, we use the linear function

$$\widehat{Q}(s,a) \stackrel{\text{def}}{=} 1\cdot\widehat{r}(s,a) + (\text{load}(s)\cdot\mathbf{w})^\top\widehat{\mathbf{\Phi}}(s,a) \qquad (12)$$

where $\widehat{r}(s,a)$ is itself an estimator that is explicitly trained to predict the execution cost $r = \Delta C/\gamma^{I(s)}$ in equation (8), and $\widehat{\mathbf{\Phi}}(s,a)$ is an estimator that is explicitly trained to predict the *reweighted future feature vector* $\widetilde{\mathbf{\Phi}}(s') \stackrel{\text{def}}{=} \frac{\text{load}(s')}{\text{load}(s)}\cdot\gamma^{I(s')-I(s)}\cdot\mathbf{\Phi}(s')$. Provided that these two estimators are unbiased, then (8) automatically holds between the estimates; $\widehat{Q}$ has no other parameters to tune.[18] In

c_t are then heavily discounted. The division not only corrects for this, but also means that Q-values tend to be large in states s that have many queries that have long been open, so these important states will dominate SARSA's least-squares estimation of Q as well as the policy gradient (10). The formal reason for the division is as follows. If s_1 were truly the start state of the Dyna solver, then of course we would have $I(s_1)=0$ and $\mathcal{O}(s_1)=\emptyset$. However, the formal definition of long-term cost $\rho(\pi)$ involves an expectation over "start" states s_1 that were drawn from the stationary distribution of the MDP under policy π, and some of these may have past queries. Dividing by $\gamma^{I(s)}$ treats these notional start states in a consistent way. It is analogous to the fact that even if a is the 51^{st} action in the sequence and thus its cost should be weighted by γ^{51}, equation (5) weights it by γ^1, in effect dividing by γ^{50} where 50 is the number of previous actions.

[17] This means samples are correlated, as well as incorrectly distributed because the path has not yet reached the stationary distribution d_π.

[18] As for $\widehat{V}$, notice that the step that tunes its parameters $\mathbf{w}$ to improve (9) (end of section 5.1) will try to bring $(\text{load}(s)\cdot\mathbf{w})^\top\mathbf{\Phi}(s)$ closer to $q = r + (\text{load}(s)\cdot\mathbf{w})^\top\widetilde{\mathbf{\Phi}}(s')$. Rearranging, this is equivalent to bringing $\mathbf{w}^\top(\text{load}(s)\cdot(\mathbf{\Phi}(s)-\widetilde{\mathbf{\Phi}}(s')))$ closer to r, which is actually just *online linear regression*—the learner tunes $\mathbf{w}$ to predict the observed execution cost r from an observed vector of feature differences, $\text{load}(s)\cdot(\mathbf{\Phi}(s)-\widetilde{\mathbf{\Phi}}(s'))$. Why? Intuitively, taking action a reduced the future cost by r, while also reducing the feature vector $\mathbf{\Phi}$. Since the weights $\mathbf{w}$ are supposed to predict *total* future

practice, observing that $\widetilde{\mathbf{\Phi}}(s) = (\frac{\Delta \text{load}}{\text{load}(s)} + 1) \cdot \gamma^{\Delta I} \cdot (\mathbf{\Phi}(s) + \Delta\mathbf{\Phi})$, we can define our prediction $\widehat{\mathbf{\Phi}}$ to take the same form, using separate estimators of the feature change vector $\Delta\mathbf{\Phi} \stackrel{\text{def}}{=} \mathbf{\Phi}(s') - \mathbf{\Phi}(s)$ and the changes to the open queries, $\Delta I \stackrel{\text{def}}{=} I(s') - I(s)$ and $\Delta\text{load} \stackrel{\text{def}}{=} \text{load}(s') - \text{load}(s)$. Linear regression estimators of r and these Δ values can consult a and the features $\mathbf{\Phi}(s)$, 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 $\mathbf{\Phi}$ includes many features. The key point is that most actions have limited ability to change the state: hence the actual feature change $\Delta\mathbf{\Phi}$ and our predicted change $\widehat{\Delta\mathbf{\Phi}}(s, a)$ will both be sparse vectors. As a result, computations like $\widehat{A}(s, a)$ (needed by (11)) are similar across nearby timesteps and can share work. It also follows that updates to linear regression weight vectors (including $\mathbf{w}$—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 widely used in both compiler and JIT optimizers. Database query optimizers use adaptive cost estimates (Deshpande et al. 2007).

Performance Portability: Performance improvements on one computer do not necessarily generalize to other architectures or all workloads. A portable performance community has formed around the idea of tuning software to be as fast as possible on a given computer. Auto-tuned numerical libraries are popular and pervasive, such as ATLAS (Whaley et al. 2001), PhiPAC (Bilmes et al. 1996) and FFTW (Frigo and Johnson 1998). Another success story is an auto-tuned sorting library (Li et al. 2004). The name of the game is adapting low-level implementation details to specific hardware. These details include loop {order, tiling, unrolling} and whether to use instruction-level parallelism.

These libraries are usually tuned *offline* by generating and benchmarking many variants. There is no reinforcement learning in the sense of a single system that gradually tunes its strategy as it runs.

Like our methods, the methods in these libraries may support multiple strategies. Like our actions, they try to dispatch a method call to the best strategy (algorithm configuration) based on features of the arguments to the call, such as matrix dimensions. However, their dispatch policy is tuned offline.

Tuning Declarative Solvers: Hutter (2009) used black-box optimization techniques to tune the parameters of solver heuristics for mixed-integer programming solvers and satisfiability solvers to optimize efficiency for specific problem workloads. This approach won several competitions, which demonstrated the effectiveness of tuning solvers to a specific workload. Hutter focused on *offline* tuning, where the solver is treated as like a *black-box* that can be repeatedly called with different parameters and problem instances. In this way his work resembles the performance portability work above, although it focuses on tuning to a workload rather than to a platform.

Database Query Optimization: Cost estimation strategies often make very weak statistical assumptions about the database contents, generally falling back to crude models based on independent uniform distributions. These choices are made for both efficiency and to serve as a good default strategy when domain knowledge is not available. However, Tzoumas et al. (2013) use probabilistic graphical models as a sophisticated approach to query selectivity estimation, which can capture richer correlations between variables in a complex filtering function (predicate).

Tzoumas et al. (2008) applied RL to query planning by training a policy to select among join and filter orders within the Eddies algorithm (Avnur and Hellerstein 2000). This is similar to how we will explore different loop order strategies inside our methods.

Database Configuration: Database configuration is traditionally an offline problem (Basu et al. 2016), which is accomplished with an expert in the loop. This is often done with the aid of database administrator tools, such as a "what-if optimizer" that predicts the utility of adding or dropping an index (Chaudhuri and Narasayya 1998). However, Basu et al. (2016) describe an RL approach to learning dynamic policies for database configuration, including which indexes to build and drop. Their approach takes database configurations as states, changes to the configuration as actions, and minimizes the long-term expected future cost under a given dynamic workload. This is rather similar to our approach to data structures, except that we usually pursue a mixed configuration and shift gradually, rather than switching abruptly from one to another.

Data structure selection and tuning: The semantics of a data structure are traditionally specified as an *abstract data type* (ADT), which 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 memos). 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 Eisner 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.

cost V from $\mathbf{\Phi}$ (after all, $\widehat{V}(s) \stackrel{\text{def}}{=} \mathbf{w}^\top(\text{load}(s) \cdot \mathbf{\Phi}(s))$), it is intuitive that they should also predict *each action*'s V reduction from its $\mathbf{\Phi}$ reduction—and that is how temporal difference learning achieves training of $\mathbf{w}$.

References

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015.

R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 261–272. ACM, 2000.

R. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Combinatorial Pattern Matching*, 2004.

D. Basu, Q. Lin, W. Chen, H. T. Vo, Z. Yuan, P. Senellart, and S. Bressan. Regularized cost-model oblivious database tuning with reinforcement learning. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XXVIII*, pages 96–132. Springer, 2016.

S. Bhatnagar, M. Ghavamzadeh, M. Lee, and R. S. Sutton. Incremental natural actor-critic algorithms. In *Advances in Neural Information Processing Systems (NIPS)*, 2007.

J. Bilmes, K. Asanović, J. Demmel, D. Lam, and C. Chin. PHiPAC: A portable, high-performance, ANSI C coding methodology and its application to matrix multiply. Technical report, U. of Tennessee, 1996.

B. Carpenter. Lazy sparse stochastic gradient descent for regularized multinomial logistic regression. Technical report, Alias-i, Inc., 2008.

S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

S. Chaudhuri and V. Narasayya. Autoadmin "what-if" index analysis utility. *ACM SIGMOD Record*, 27(2):367–378, 1998.

A. Colmerauer and P. Roussel. The birth of Prolog. In *History of programming languages—II*, pages 331–367, 1996.

A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends® in Databases*, 1(1):1–140, 2007.

A. Dunlop, N. Bodenstab, and B. Roark. Reducing the grammar constant: an analysis of CYK parsing efficiency. Technical Report CSLU-2010-02, Oregon Health and Science University, 2010.

J. Eastep, D. Wingate, M. D. Santambrogio, and A. Agarwal. SmartLocks: Lock acquisition scheduling for self-aware synchronization. In *Proc. of the International Conference on Autonomic Computing*, 2010.

J. Eisner. Dyna: A *non*-probabilistic programming language for probabilistic AI. NIPS*2008 Workshop on Probabilistic Programming, 2008.

J. Eisner and J. Blatz. Program transformations for optimization of parsing algorithms and other weighted logic programs. In S. Wintner, editor, *Proc. of the Conference on Formal Grammar*, pages 45–85, 2007.

J. Eisner and N. W. Filardo. Dyna: Extending Datalog for modern AI. In O. de Moor, G. Gottlob, T. Furche, and A. Sellers, editors, *Datalog Reloaded*, volume 6702 of *Lecture Notes in Computer Science*, pages 181–220. Springer, 2011. Longer version available at `http://cs.jhu.edu/~jason/papers/#eisner-filardo-2011`.

J. Eisner, E. Goldlust, and N. A. Smith. Compiling comp ling: Weighted dynamic programming and the Dyna language. In *Proc. of the Joint Conference on Human Language Technology Conference and Empirical Methods in Natural Language Processing (HLT/EMNLP)*, pages 281–290, October 2005.

G. Elidan, I. McGraw, and D. Koller. Residual belief propagation: Informed scheduling for asynchronous message passing. In *Proc. of UAI*, 2006.

P. M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3), 1994.

N. W. Filardo and J. Eisner. A flexible solver for finite arithmetic circuits. In *International Conference on Logic Programming*, volume 17 of *Leibniz International Proc. in Informatics*, pages 425–438, 2012.

N. W. Filardo and J. Eisner. Set-at-a-time solving in weighted logic programs. 2017a. URL `http://www.ietfng.org/nwf/_downloads/2017-set.pdf`. Under review.

N. W. Filardo and J. Eisner. Default reasoning in weighted logic programs. 2017b. URL `http://www.ietfng.org/nwf/_downloads/2017-default.pdf`. Under review.

P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *Analysis of Algorithms*, pages 127–146, 2007.

M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, volume 3, 1998.

A. Garlapati, A. Raghunathan, V. Nagarajan, and B. Ravindran. A reinforcement learning approach to online learning of decision trees. *CoRR*, abs/1507.06923, 2015.

A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves. Memory-efficient backpropagation through time. *Computing Research Repository (CoRR)*, abs/1606.03401, 2016.

H. He, P. Mineiro, and N. Karampatziakis. Active information acquisition. *CoRR*, abs/1602.02181, 2016.

F. Hutter. *Automated Configuration of Algorithms for Solving Hard Computational Problems*. PhD thesis, University of British Columbia, 2009.

R. Kowalski. Predicate logic as programming language. Memo 70, Edinburgh University, 1974.

X. Li, M. J. Garzarán, and D. Padua. A dynamically tuned sorting library. In *International Symposium on Code Generation and Optimization*, 2004.

N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *USENIX File & Storage Technologies Conference*, 2003.

M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. of the Design Automation Conference*, pages 530–535. ACM, 2001.

J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

M. Schmidt. *Graphical Model Structure Learning with ℓ_1-Regularization*. PhD thesis, University of British Columbia, 2010.

M. Schmidt and K. Murphy. Convex structure learning in log-linear models: Beyond pairwise potentials. In *Proc. of the Workshop on Artificial Intelligence and Statistics (AI-Stats)*, pages 709–716, 2010.

E. Strubell, L. Vilnis, K. Silverstein, and A. McCallum. Learning dynamic feature selection for fast sequential prediction. In *Proc. of the Conference of the Association for Computational Linguistics (ACL)*, 2015.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. 2017. URL `http://incompleteideas.net/sutton/book/the-book-2nd.html`. Second edition. Preprint.

R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1057–1063, 2000.

Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *Computing Research Repository (CoRR)*, abs/1605.02688, 2016.

K. Tzoumas, T. K. Sellis, and C. S. Jensen. A reinforcement learning approach for adaptive query processing. Technical report, Aalborg University, DB Tech Reports, 2008.

K. Tzoumas, A. Deshpande, and C. S. Jensen. Efficiently adapting graphical models for selectivity estimation. *The International Journal on Very Large Data Bases*, 22(1):3–27, 2013. doi: 10.1007/s00778-012-0293-7.

J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

H. van Seijen, H. van Hasselt, S. Whiteson, and M. Wiering. A theoretical and empirical analysis of Expected Sarsa. In *Proc. of the IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (APDRL)*, 2009.

R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1), 2001.

G. Zweig and M. Padmanabhan. Exact alpha-beta computation in logarithmic space with application to map word graph construction. In *International Conference on Spoken Language Processing (ICSLP)*, 2000.