Adaptive Self-Tuning Dyna

Tim Vieira
tim.f.vieira@gmail.com

1 Introduction

Declarative programming is a paradigm that allows programmers to specify *what* they want to compute, leaving *how* it will be computed to a solver. The declarative specification provides a level of indirection between specification and execution, which gives a solver the flexibility to choose between many options for execution. In this proposal, I will describe opportunities for adaptive execution in the Dyna language's solver. I propose a reinforcement learning approach for adaptively tuning many components of Dyna's flexible solver to jointly maximize efficiency for a given workload.

Human programmers manage to make good choices when designing their code. We hope to design policies that encompass human-designed heuristics. We also hope that the more exhaustive experimentation that reinforcement learning does will find good policies without too much overhead. Adaptivity is especially important for software that will run under a variety of unknown workloads, where no fixed policy works well.

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 with rules (equational schemata). To get a sense for the language, we'll look at a classic example: multi-source shortest path in a directed graph,

```
path(S,T) min= path(S,U) + cost(U,T).

% edge costs
cost("bal","phi") = 1.
cost("phi","nyp") = 2.
...
```

As you can see, a Dyna program is a collection of **rules**, which are *templates* for equations that relate the *values* of *items* in the graph mathematically. Theses rules include variables, which are denoted syntactically as upper case symbols. Rules are expanded by pattern matching against other items in the program. Semantically, Dyna is simply an interface for querying for value assignments to items that satisfy the equations.

How do equation templates work? In our example, the first rule says that we should, by some mechanism, be adding the values of all known `path(S,U)` and `cost(U,T)` items. Since the variables `S` and `T` appear in the head of the rule, we will loop over remaining variables (just `U`, in this case). This gives rise to multiple instantiations of the body, which will be *aggregated* over by a designated associative-commutative operator, in this case `min`. When a variable, like `U`, is used in multiple items in the body, it serves as a *filter*.

Dyna programs support general mathematical expressions. The following example defines a multi-layer neural network with edge weights `w(J,I)` and sigmoid nonlinearities. It also defines the training objective `error`: the sum of squared error at `target` nodes.[1]

```
σ(X) = 1/(1+exp(-X)).
output(I) = σ(input(I)).
input(I) += output(J) * w(J,I).
error += (output(I) - target(I))**2.
```

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 *lazily* unrolls the graph to answer a query, e.g.,

[1] Dyna supports automatic differentiation to enable gradient-based optimization and sensitivity analysis.

`fib(13)`. To support inference in cyclic graphs, the solver repeatedly applies rules until they reach a fixed point. Not all Dyna programs terminate. A fixed-point may not exist and there may be more than one. Some programs only terminate numerically, such as the geometric series,

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

1.2 The Dyna solver

Traditionally, computation graphs are updated in one of two extreme modes: (1) **backward chaining**, where we work backwards from a specific query item towards inputs to find its value. (2) **forward chaining**, which propagates information from inputs towards all other nodes.

Backward chaining works well for infinite circuits because it only explores the necessary regions of the graph, but it fails to handle cyclic dependencies. Back chaining fails to efficiently support updates to the inputs because it must flush all memos that might depend on the inputs that changed. On the other hand, forward chaining lives to propagate incremental changes. Unfortunately, forward chaining has a large memory burden because it memoizes all intermediate values and it fails to terminate on infinite programs because it will explore *everything*.

The Dyna solver (Filardo and Eisner, 2012) is based on a **mixed-chaining** strategy, where an agenda-based algorithm uses forward chaining to update existing memos and backward chaining to create memos. The mixed-chaining solver allows memos to be flushed or created at any point at the whims of a **memoization policy**, while lazily maintaining the consistency of existing memos. At any point the solver can (1) answer a query, which uses existing memos to make computation efficient and (2) support updates to the inputs, which will incrementally update existing memos with the possibility of (a) marking existing memos as invalid, (b) eagerly pushing the updates through the graph, or (c) something in between.

Since the solver is an agenda-based algorithm, it supports flexible *computation order*; the decisions about "what to work on" are controlled by a **prioritization policy**. Picking a good order is important because it controls how we explore the graph. A bad policy will result in unnecessary repropagation of updates and slow convergence, especially around cycles.

The Dyna solver was designed with machine learning in mind. We will discuss how we plan to learn adaptive memoization and prioritization policies with reinforcement learning. We will use the same technique to make lower-level decisions, including query and update planning as well as data structure selection.

1.3 Adaptive tuning, more formally

We formalize the goal of adaptive tuning as an online optimization problem. For a given Dyna program, a **workload** $p_{\mathcal{W}}$ is a distribution over a *stream* of queries and updates, $w_1, w_2, \ldots, w_n \sim p_{\mathcal{W}}$. Formally, we'd like minimize the average total time (makespan) for the solver to process an entire stream from this workload,

$$\text{minimize } \mathbb{E}\left[\sum_{i=1}^{n} c_i\right],$$

where $(s_{i+1}, c_i) \sim p_{\mathcal{S}}(\cdot|\pi_i, s_i, w_i)$ indicates how the internal state of the Dyna solver s_i evolves as it processes the *work item* w_i (a query or update) with the policy π_i and c_i measures the cost (runtime) of the processing at that step. We assume that the solver, by construction, responds appropriately to the work item, i.e., gives correct answers to queries. We assume that the work load is processed sequentially.[2]

Note that the solver has some interesting flexibility in how it processes the stream. Consider two extremes: (1) the solver can eagerly process all updates so that queries are efficient. (2) the solver can buffer the updates and only apply them when they are need to answer a query.

We'd like the solver to adapt its internal policies to make processing this stream as fast as possible. This includes adapting to any *nuances*,

[2] In future, we may want to allow the solver to respond to work items out of order via callbacks. In such settings, we may want to formulate the objective differently to account for latency or other scheduling considerations.

e.g., if we appear to be performing lots of updates, maybe we want to preemptively clear dependent memos in bulk, which is more efficient than propagating changes one at a time.

Optimization is fueled by intermediate costs that are measured after each execution step. These runtimes are given as feedback to the optimization algorithm, which adapts the sequence of policies to minimize future costs. Optimization is constrained to search for a policy in some policy space. We'll describe specific policy spaces later in this document.

Now that we've formalized adaptive tuning problem as an optimization problem, we're free to optimize in any way we like. For example, **black-box optimization** techniques are methods that attempt to optimize functions given only access to stochastic function evaluations (no gradients).[3] Black-box optimization on policy parameters is often competitive with sophisticated RL algorithms in low-dimensional problems, but less so in higher dimensions. Thus, black-box optimization will make a good baseline method for my future experiments.

2 Adaptive tuning framework

Abstractly, the execution of an algorithm is a sequential decision process. 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. The system executes the action and transitions, $(s_{t+1}, r_t) \sim p(\cdot|s_t, a_t)$, to the next state s_{t+1} and observes a scalar reward signal r_t. This process is called a Markov decision processes (MDP). The agent seeks to optimize *long-term* total reward,

$$\rho_T(\pi) = \mathbb{E}\left[\sum_{t=1}^{T} r_t\right]$$

Or, the discounted reward,

$$\rho_\gamma(\pi) = \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^t r_t\right]$$

where there parameter $0<\gamma<1$ ensures a convergent sum and controls the extent to which the agent cares about future.

The optimization is carried out by *interacting* with the environment because the transition and rewards are not known a priori; this protocol is known as reinforcement learning (RL). Programs written in this framework are known as **adaptive programs** (Pinto, 2015) (although this term is not widely used). A similar framework is described in Chang et al. (2016).

The difficulties of learning in the RL setting are primarily • **Credit assignment problem**: Since reward is delayed, determining which actions are responsible for subsequent rewards or punishments is difficult and may require multiple rounds of experimentation. • **Exploration-exploitation dilemma**: Since the correct action is not given to the agent directly, it must explore the environment to find good actions to take. Additionally, the agent needs to learn to transition to (and remain in) high-reward states.

2.1 Definitions

Before continuing on to algorithms, we first define a few important concepts. The **action-value function** for a policy π, is a map from all states-action pairs to their expected future reward for following π from then on,

$$Q_\pi(s,a) = \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^t\, r_t \;\middle|\; s_0 = s, a_0 = a\right]$$

We also define two similar functions: the **value function**, $V_\pi(s) = \sum_a Q_\pi(s,a)\pi(a|s)$, and **advantage function**, $A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s)$. Note that $\rho(\pi) = V_\pi(s_0)$, assuming (without loss of generality) that s_0 is the fixed initial state of the system. Let d_π denote the stationary distribution over states of an agent following π. We abuse notation and write $(s,a) \sim d_\pi$ as short hand for $s \sim d_\pi$ and $a \sim \pi(\,\cdot\,|s)$.

Many algorithms in RL are based on approximating of a subset of Q, V, A, or π with a parametric function approximator. This crucial to

[3] Examples of black-box methods: • Approximate gradient methods: finite-difference (Kiefer and Wolfowitz, 1952) and simultaneous perturbation (Spall, 2003, Ch. 7) • Evolutionary methods: CMA-ES (Hansen, 2006), simulated annealing (Kirkpatrick et al., 1983). • Local search methods: Nelder-Mead Simplex (Nelder and Mead, 1965), stochastic local search (e.g. Hutter (2009), Spall (2003, Ch. 2)) • Model-based optimization: Bayesian optimization (e.g., Hutter (2009), Snoek et al. (2012))

scaling RL to large state spaces, which requires the agent to generalize across states because it will never get to visit all of them. We'll discuss one algorithm which learns a parametric policy, but also leverages approximations to Q and V to speed up learning.

2.2 Policy gradient

Policy gradient algorithms (Williams, 1992; Sutton et al., 1999) are an important family of algorithms in RL because they are guaranteed to converge to local optimum when function approximation is used to represent the policy, including nonlinear approximators.

Consider a stochastic policy π_θ, which is continuously differentiable in θ. The gradient of expected reward $\rho(\theta)$ is[4]

$$\nabla_\theta \rho(\theta) = \mathop{\mathbb{E}}_{(s,a)\sim d_\pi}\Big[Q_\pi(s,a)\,\nabla_\theta \log \pi(a|s)\Big]$$

Now, we'd like to get a Monte Carlo estimator for $\nabla_\theta \rho$ by sampling (s,a) and computing the value in the body of the expectation, but $Q_\pi(s,a)$ is not available. So we'll have estimate it. We'll describe a few options for how to do this effectively in the remainder of this section.

In episodic MDPs, where all paths reach a terminal state in finite time, we can take a sample path $[s_0\, a_0\, r_0 \ldots s_T\, a_T\, r_T]$ and estimate Q_π as the sum of future rewards,

$$Q^{\mathrm{R}}_\pi(s_t,a_t) = \sum_{j=t}^{T} r_j\, \gamma^{j-t},$$

to form a simple Monte Carlo estimator, known as REINFORCE (Williams, 1992).

2.3 Influence estimates

REINFORCE captures the independence of *past* rewards on future actions. Can we leverage information about about the independence of *future* rewards on past actions?

The REINFORCE gradient estimate simply tries to increase the probabilities of trajectories that gave rise to higher rewards. This means that *any* action along the path gets credited for the sum of all rewards that came after it, including rewards they didn't have anything to do with!

For example, suppose that the MDP is secretly a contextual bandit, i.e., each state is sampled independently. In this case, future rewards beyond the immediate reward are independent of all past actions. Unfortunately, policy gradient doesn't know that so it (noisily) gives a_t credit for future rewards. Of course, this "credit" washes out as we average over many samples. However, we can improve the variance of the estimator, while maintaining unbiasedness, if we are able identify which rewards depend on which actions.

Influence estimates (Pinto, 2015) are a simple framework for doing just that. The way they work is by breaking up the total future reward into sub-totals that reflect the causal structure. The influence relation I is a function, $I(s_t,a_t,r_j) \in \{0,1\}$, which returns 0 when we know that the reward r_j, $j > t$, is statistically independent of action a_t given s_t. False positives are allowed, but false negatives will introduce bias. The function I allows a user or an automated analysis system to make conservative assertions about conditional independence relationships among action-reward pairs. The more independence assertions made, the lower the variance of the gradient estimator.

$$Q^{\mathrm{I}}_\pi(s_t,a_t) = \sum_{j=t}^{T} I(s_t,a_t,r_j)\, r_j\, \gamma^{j-t}.$$

Of course, temporal discounting means rewards that are far away in time contribute very little to the gradient signal at an action even if that action was the only thing responsible for it. Pinto (2015) suggest using the number of *relevant* actions $D(s_t,a_t,r_j)$ instead of the observed time difference for discounting. This idea is intuitive, but introduces bias (which is often beneficial),

$$Q^{\mathrm{D}}_\pi(s_t,a_t) = \sum_{j=t}^{T} I(s_t,a_t,r_j)\, r_j\, \gamma^{D(s_t,a_t,r_j)}.$$

[4] We can safely push the gradient through the expectation under mild conditions. Interestingly, d_π and Q_π, which depend on θ, do not complicate to the gradient.

I will explore modifications of the influence-based estimators that will support the non-episodic setting. I will describe causal relationships we plan to leverage for adaptively tuning Dyna in section 3.

2.4 Incremental algorithm

We can get fully-incremental algorithm by approximating $Q_\pi(s,a)$ with a function estimator. If we do this in an arbitrary manner, then we could end up with a biased estimator. However, if we use a *compatible parameterization*, $Q^w_\pi(s,a) = w^\top \nabla_\theta \log \pi(s|a)$ and pick w to minimize mean squared error,

$$\mathcal{E}_\pi(w) = \frac{1}{2} \mathop{\mathbb{E}}_{(s,a)\sim d_\pi} \left[\left(Q_\pi(s,a) - Q^w_\pi(s,a) \right)^2 \right].$$

then will get an unbiased Monte Carlo estimator of $\nabla_\theta \rho$ by using Q^w_π in place of Q_π. Of course, this is not practical because minimizing $\mathcal{E}_\pi$ before each gradient estimate would be very costly.

A clever way to minimize $\mathcal{E}_\pi$ without needing to estimate $Q_\pi(s,a)$—the very thing we were trying to avoid—is to use the *temporal difference error* to estimate $\nabla_w \mathcal{E}_\pi$, which says that the estimator at time t should agree *with its own predictions* at the next time step $t+1$,

$$\delta_t = \Big(r_{t+1} + \gamma Q^w_\pi(s_{t+1}, a_{t+1}) \Big) - Q^w_\pi(s_t, a_t)$$

$$\nabla_w \mathcal{E}_\pi(w) = -\delta_t \nabla_\theta \log \pi(s|a)$$

We can optimize w and θ at the same time by stochastic gradient using their respective gradient estimates.[5] This gives us an incremental algorithm, which doesn't requiring sampling entire episodes in order to compute a gradient estimate. It's even safe to use consecutive samples from executing the policy. This process converges to a local optima of ρ (Bhatnagar et al., 2007).

[5] Optimizing w is much easier that θ, because $\mathcal{E}_\pi$ is convex in w. However, because we are updating θ at the same time, $\mathcal{E}_\pi$ is actually nonstationary.

2.5 Baseline functions[6]

We can improve variance significantly using a state-dependent baseline function,

$$\nabla_\theta \rho(\theta) = \mathop{\mathbb{E}}_{(s,a)\sim d_\pi} \left[\Big(Q_\pi(s,a) - b(s) \Big) \nabla_\theta \log \pi(a|s) \right]$$

where $b(s)$ is an arbitrary function of the current state that is independent of θ. We are free to use any approximating family we like to represent b, e.g., neural networks. We are also free to tune b however we like without introducing bias into gradient estimation. A popular choice is direct variance minimization by stochastic descent on the variance objective at the same time as optimization on θ.

2.6 Counterfactuals

RL algorithms generally do not assume the ability to *roll-back* to a previously encountered state and try alternative actions. We refer to evaluating alternative actions as a *counterfactual*: How does expected reward change if we vary this one action?

We believe that, in our setting, there are a few special cases where we can cheaply and accurately estimate the advantage of taking each actions in a given state. These estimate will be noisy and will assume other sources of uncertainty stay constant. This gives us *paired* feedback for multiple actions. Statistical methods based on paired sample generally have lower sample complexity. In section 3, we'll describe how we'll obtain these estimates more concretely.

2.7 Remarks

The role of b and Q: Neither b nor Q estimates are needed in order to execute the policy, they are simply instrumental to learning. In effect, the difference of these estimates is an estimate of the advantage function $A_\pi(s,a)$. Intuitively, the advantage function is a counterfactual: What if we change the action taken in state s and follow by current policy from then on? We might consider more expensive means of approximating A_π that we would for π, since it serves

[6] Baseline functions are more of a detail, not essential to our discussion.

to teach π to act and won't effect the runtime of the policy after learning stops. Of course, we might never turn off learning.

Black-box methods vs. policy gradient: The policy gradient method we described gives us a gradient estimate, which enables search in high-dimensional policy spaces. Unlike black-box methods, policy gradient leverages the temporal structure in the MDP, which often improves gradient estimation. We showed how to further improve the basic REINFORCE estimator by leveraging additional independence information expressed via influence relations. One benefit of some black-box methods is that they don't assume that the ρ is differentiable, which would allow deterministic policies families.

3 Adaptively Tuning Dyna

In this section, I will describe four of the important decision types that our adaptive policies will learn to make. Each decision type has a dedicated subsection, which • characterizes the problem, • discusses the impact of making the decisions well, and • describes the relevant information that a policy can use as features.

Additionally, I will also discuss the relevant influence structure (section 2.3) that we plan to leverage in order to make RL training efficient. I will also describe how we view the Dyna solver as an MDP and the policy spaces we plan to explore in this work.

3.1 The Dyna Solver MDP

In section 2, I described an abstract framework for adaptive programming, which uses Markov Decisions Processes, for sequential decision making. In this subsection, will be a little more concrete about what the specific MDP looks like.

State space The state space of our solver is astronomically large; it includes the memo table, agenda, program source, existing indexes, and any bookkeeping information.

Action space The action space depends on the type of decision the policy is making. For memoization, the solver may have just computed some value and then it asks the policy whether or not to memoize it. For prioritization, when the solver tries to schedule an update on the agenda it will ask the policy to assign a priority to the update. More details will be given in later subsections for each decision type.

Transition function The transition function is very complex as it implements a step of the actual solver. For a more complete description of the solver, I refer the reader to Filardo and Eisner (2012). Intuitively, the transition function is "whatever happened between calls to the policy." Generally, the transitions of the solver are deterministic. The solver can query the policy for different types of action in whatever order they arise during execution.

Importantly, however, there are also transitions that are caused by the work items (section 1.3). Unlike the queries and updates made by the solver, these external queries and updates are *uncertain* because they come from the outside world $p_{\mathcal{W}}$. Similarly, the initial inputs to the program are conceptually just uncertain updates to an initially empty program.

Some of the details about exactly when the policy is queried are going to be explored as part of this proposal. Especially in the case of memoization: Do we get have the opportunity to flush memos every time we access them? Do we asynchronously loop over the memo table looking for stuff that isn't being used? Are there natural triggers for identifying items we might want to flush?

Reward function Along each transition, we execute the action given by policy, which takes some amount of time. That amount of time is the immediate cost (negative reward) of performing the chosen action. As usual, some actions, such as memoizing a query, will only receive a *delayed* reward for speeding-up later actions.

Policy space We think that *regression trees* are a nice family of policies since they can *induce* complex features from a space of simpler ones. Regression trees can also handle numerical and categorical features. The internal nodes of the regression tree can include numerical comparison operators, linear functions, and Boolean functions, which make regression trees capable of

expressing rich nonlinear functions. Additionally, regression trees are fast to evaluate and can be interpreted by humans. We also think that regression trees are similar to the space of policies that a human programmer would consider for our types of problems.

Unfortunately, regression trees aren't continuously differentiable stochastic policies, like policy gradient requires. I'm interested in developing extensions that will support this type of training, but I'm not sure how to do it yet.

In the meantime, we can use Boltzmann distributions with linear feature functions,

$$\pi_\theta(a|s) \propto \exp\left(\theta^\top f(s,a)\right),$$

where $f(s,a)$ are features of the state-action pair. We can get some of the benefit of feature induction by considering a huge a feature space and regularizing θ with an active set method so that only active features need to be computed.[7]

We can also use non-linear scores given by neural networks that maps state-action pairs to a scalar value, which we can use in a distribution,

$$\pi_\theta(a|s) \propto \exp\left(\mathrm{NN}_\theta(s,a)\right).$$

We strongly suspect that the most effective policies will be very simple, since humans appear to use very simple heuristics for solving these problems.

3.2 Prioritization

A good prioritization policy must understand the relationships among terms: What types of items send messages to each other? Roughly, this is a question about the topology of the computation graph. Performing updates in a topological order is ideal because it avoids repropagation. However, in Dyna, we don't generally know the topological order because we are only discovering the graph as we go. When the graph is cyclic, no topological order exists. Also, the graph structure might even change depending on the value of an item.

[7] In Vieira et al. (2016), we considered all possible n-grams as features in a sequence tagger problem. We used a tree-structured group lasso penalty (Nelakanti et al., 2013) with an active set method to learn a good set of features from the enormous space of possible n-grams.

Some topological understanding of a Dyna program can be derived with static program analysis. However, dynamic characterization is much more effective since more information is available. For example, we don't know based on the Dyna rule for constituency parsing,

```
phrase(X,I,K) max= phrase(Y,I,J)
      * phrase(Z,J,K) * rule(X,Y,Z).
```

that we might want to prioritize by item width `K-I`, as in the CKY algorithm, because shortest-width first gives rise to a topological order.

To illustrate why Dyna can't recover this strategy by static analysis, we'll reconsider our `K-I` idea. It turns out that `K-I` is only the "width" because of a convention for how we specify the input. We could have specified the input as a finite-state machine and used the same program to parse it. In that case `I,J,K` are states of the finite-state machine, which can be named arbitrarily, e.g. with strings. Thus, `K-I` is meaningless under that kind of input. To make matters worse, Dyna doesn't even know if the finite-state machine encoding the input is acyclic; the same rule can parse *any* finite-state machine.

Suppose we extend our program to support unary rules, we now have to worry about cycles.

```
phrase(X,I,K) max= phrase(Y,I,K) * rule(X,Y).
```

A good strategy in cyclic programs is to propagate messages within a strongly connected component (SCC) until (numerical) convergence before moving to the next SCC. In this example, items that cover the same span `(I,K)` form strongly connected component.

The policy that I will develop will derive features from the following sources:[8]

- Name-based: We can derive features from the name of an item since it often encodes useful information, e.g., the width of `phrase(X,I,K)` is a linear function of `I` and `K`.
- Value-based: An item's current value or

[8] In prior work, we explored many of these feature sources for learning prioritization heuristics, which were specifically for constituency parsing.

change in value is a useful signal. In parsing, exploring the highest-valued items first gives rise to the uniform-cost search strategy, which is analogous to Dijkstra's shortest path algorithm. Of course, not all items have numeric values; such items must look to the other sources for features or use these types of values categorically (similar to treating values as names).

- Temporal: We can use features based on when the item was last updated or looked-up to recover strategies like "first in first out" and to prevent update starvation, e.g., by encouraging the solver to process items that have been on the agenda for a long time.

- Topological: We'd like to use features that "sketch" the shape of the graph, including what is pending on the agenda, to help approximate a topological order. These features might be computed by static analysis, such as coarse-grained analysis (`word` feeds into `phrase`). However, dynamic analysis may be more informative. We'd also like features which help us find strongly connected components.

- User-defined: We also make the priority function and its features user-definable in Dyna, e.g., `$priority(phrase(X,I,K)) = K-I`.

Since items of multiple types are scheduled on the same agenda, the prioritization policy must *calibrate* features to be on a comparable scale, especially for value-based features. This means that a scaling factor and possibly a nonlinear transformation may be necessary.

3.3 Memoization & Indexing

Creating memos *saves* work when they are reused. However, it is not the case that we want to memoize everything because maintaining memos *adds* work.[9,10] Failing to memoize at certain points in a computation can drastically increase runtime. For example, computing the n^{th} Fibonacci number without memoization is exponential in n, instead of linear. Carefully chosen memo points can save *lots* of work because they stop back chaining from recursing further down the graph.

Indexing the memo table allows the solver to efficiently look stuff up (query). We regard deciding what to index and deciding what to memoize as the same problem.

An interesting example of a clever memoization policy is "memory-efficient backpropagation" (Zweig and Padmanabhan, 2000; Gruslys et al., 2016), where cleverly placed checkpoints (memos) in the forward pass get reused by the backward pass. Checkpoint locations are chosen to reduce space complexity with little increase in runtime. For example, $\mathcal{O}(\sqrt{n})$ space with $\mathcal{O}(2n)$ time, or $\mathcal{O}(\log n)$ space with $\mathcal{O}(n \log n)$ time.

Memoization policies are a prime candidate for leveraging influence information (section 2.3) because we understand the causal structure of the memoization process.

Consider a specific item in the graph. At some point during solving, call it time t, we computed its value and decided whether or not to memoize it (action $a_t \in \{\mathsf{memo}, \mathsf{flush}\}$). At a number of future times $\{q_1, q_2, \ldots\}$ we query the item. Similarly, we update the item at future times $\{u_1, u_2, \ldots\}$. We know that the rewards of *all other times* are not influence by this action, $I(s_t, a_t, r_k) = 0$ for $k \notin (\{q_i\} \cup \{u_j\})$. This is a already a huge improvement over the REINFORCE estimate, but there are a few more improvements. If $a_t = \mathsf{flush}$ then the future update u_i has nothing to do with past action, $I(s_t, a_t, r_{u_i}) = 0$. If t is *covered* by another memo, which stops the solver from recursing down to t, then we can assert another independence relationship.

We now discuss some ideas for counterfactual estimates for queries.[11] For counterfactual feedback, we consider what the solver *would have done* in each case. For queries, the benefit of switching a_t from flush to memo is fast to compute because we have already searched the subgraph

[9] Additionally, looking up a memo may require memory access, which can be slower than recomputation in some cases, e.g., floating point functions.

[10] We will ignore concerns about managing memory consumption. For now, we focus on the cost of maintenance.

[11] I have not thought through the details of how to get counterfactuals for updates.

below the item to satisfy the query. Now, we just have to consider the benefit of stopping earlier. The other direction, however, is slower to compute because we need perform the search which was suppressed by the original memo.[12]

I will use a similar policy space as for prioritization, except here we are making a classification decision. I'll use name-based, temporal, topological, and user-defined features like I described before. In this case, topological features look at what's been memoized near an item as an estimate of how far it will have to back-chain far to compute the value again. We can look at how long it took to execute last time, this gives us simple rules of thumb like "if it takes longer than x milliseconds to compute, store it." Temporal features can help recover strategies like least-recently used. Frequency features can recover strategies like "If we keep asking about something, better remember the answer." In many of these cases, features will be computed at a coarse-grained level, e.g., frequency statistics at the type level instead of item level.

3.4 Query and update planning

In order to run forward or backward chaining, the solver has to generate subroutines that can ground out the rules in a number of ways. These are subroutines are called *plans*.

For forward chaining, we need to generate an *update plan* for each item in the body that might get updated.[13] In each of these routines, the updated item *drives* the enumeration of edges that match the driver. In other words, this routine creates an iterator for the set of *outgoing* edges from the driver. Backward chaining is similar, except it searches for *incoming* edges matching the head of the rule (*query plan*).

Since Dyna allows a number of infinite relations, like most math operations, certain *queries modes* are not allowed because enumerating them would be impossible. For example, we disallow the solver to ask the addition relation (`Z==X+Y`) to enumerate all of its triples, but it is fair to ask it to find `X` such that `X+1=4`, since that's just subtraction. More generally, all relations expressed in the Dyna program have a set of *support modes*.

The planner, which assembles these *plans*, has to find a routine that only uses supported modes. In general, there may be exponentially many allowed plans. Some plans will run much faster than others. The efficiency of these routines is determined primarily by the *join order*, which is roughly what order that we loop over the results of each passenger query. Some loop orders result in more probes than others and fewer failed pattern matches. Additionally, a good plan should leverage existing indexes on the memo table.

A quick example from my own experience: In carrying out the experiments for Vieira and Eisner (2017), I learned from Dunlop et al. (2010) that CKY can be made 7-9x faster with an unexpected loop order: a sparse loop on left children instead of a cross-product of sparse left *and* sparse right right children. Even though the left-child loop does more probing, it has much better memory access patterns due to cache locality. The cross product strategy is the strategy taught in NLP courses. This stuff matters!

In this case, there are no interactions between the plan chosen and the rest of the environment. Thus, plan optimization is a contextual bandit problem. The context is very rich as it includes the state of the memo table and what indexes are available. To start, I will train this system with policy gradient on the immediate reward, which falls out of the influence estimation framework. However, there are likely better algorithms for contextual bandit learning, which has been an active area of research in recent years.

We can leverage paired samples (counterfactuals) by running several plans in the same state. We can do this because we can control the effects on system. Another idea is to run several plans in parallel threads and kill versions that were slower than the fastest thread or allowing them to complete to gather statistics.[14]

[12] We may be able to do this with bookkeeping information that computes this incrementally.

[13] By default, Dyna assumes that all items can be updated. It provides an annotation scheme for declaring that an items of a given type are *constant*. Math operations, like addition and exponentiation, are constant.

[14] Of course, we can also devise a method that learns from truncated observations. For example, Hutter et al. (2013) developed a method for learning from truncated

3.5 Data structure selection

Data structure selection and configuration is crucial to good performance. These decisions depend strongly on query and update workloads, since the workload dictates the dynamic what operations are done to the data structure (how will the data change over time?) and well as the data size and sparsity.

Selecting aggregator classes: There are also many choices for how to implement aggregators. This choice depends strongly on the types of updates an aggregator expects to receive. For example, the max aggregator might want to store its aggregands in a heap data structure, if it expects that aggregands might be retracted or their values might decrease.[15] Another example is `+=`, which can be implemented with a number of methods, including Kahan summation or a Fenwick tree (Fenwick, 1994).

Using a stochastic policy here means that we will randomize which aggregator data structure gets used for each item. Based on feedback from each aggregator instance, the policy adapts the rate it uses each data structure, favoring the ones that seem to work best in a given context.

These decisions are made once for each item when the aggregator is initially allocated. Some data structures might have parameters that need tuning (e.g., initial sizes, growth rates, miscellaneous thresholds). We will randomize these parameters as well in order to learn.

Memo storage choices: Dyna needs to determine how to store its memo table. For example, whether or not to sort on specific columns. Storing sorted data improves the efficiency of intersection and join operations, e.g., Baeza-Yates (2004) intersection. Another option that we'll consider is storing data in a trie-like data structure. In that case, we have to choose the order, e.g., `Y->X->Z` is best for our CKY example.

In the running CKY example, we want to sort `rule(X,Y,Z)` on `Y` and store `(Y,Z,value)` densely to maximize cache locality (section 3.4).

I suspect that we can get good counterfactual estimate for sorted vs. unsorted by estimating the number of comparison operations each setting would have made. Estimating benefits in terms of cache performance will require a cost model or actual execution of different versions. That level of optimization might not be needed.

3.6 The free-choice operator[16]

Most of Dyna's auto-tuning opportunities are behind the scenes and don't affect the semantics of the program. Dyna has a user-facing construct called the free choice operator (`a ?= (:double X).`), where the programmer explicitly tells Dyna that it is free to choose among options of a given type when assigning the value to the item. To choose values meaningfully, the interpreter is told to optimize an accuracy measure (in addition to runtime). In many cases, we expect that the performance measure will be defined as a continuous function of other parameters, which will enable gradient-based optimization. If parameters are discrete we can tune them with policy gradient (where we sample values from a stochastic policy defined over the options). To fit it into the RL framework, we simply augment the reward function with the user's accuracy measure. Accuracy might only be measured on a complete graph, which means that we don't get as many useful intermediate rewards from accuracy as we do from runtime. This might make learning harder. We can still leverage influence estimation to improve credit assignment in policy gradient estimation by using the structure of the computation graph to see which items depend on the `?=` items.[17]

4 Related work

In NLP and other applied machine learning fields, it is not uncommon for implementation details to constitute an entire publication! Some examples: Efficient inference in models with certain structures (Dunlop et al., 2011; Saragawi, 2006; Lopez,

observations in Bayesian optimization.

[15] In Vieira and Eisner (2017), our learning algorithm flipped pruning decisions, which cause many aggregands to be retracted then added again or the other way around.

[16] This is a speculative section.

[17] This is similar to stochastic computation graphs, which extend backprop to support differentiation through sampling-based expectations (Schulman et al., 2015).

2007; DeNero et al., 2009), memory-efficient backpropagation in recurrent neural networks (Chen et al., 2016; Gruslys et al., 2016). Many of these publications are based on combinations of standard tricks (e.g., loop-order optimization and cache optimization of a data structure, finite-state machine minimization, common expressions elimination, magic sets transformation). Some implementation tricks passed into folklore before they were ever published. Not all tricks are widely known or even known to the people who would benefit. Most tricks are tedious and error prone to apply manually.

Portable Performance: 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. There are many success stories with auto-tuned for number crunching libraries, such as ATLAS (Whaley et al., 2001), PhiPAC (Bilmes et al., 1996) and FFTW (Frigo and Johnson, 1998). These libraries are pervasive: you've probably used them many times without knowing it! Another success story is tuning sorting libraries (Li et al., 2004). The name of the game is adapting low-level implementation details to specific hardware.[18] These details include loop {order, tiling, unrolling} and whether to use instruction-level parallelism.

Databases are home to many interesting adaptive strategies since they have many internal components that require cleverness. For example, **query planning** seeks to optimize join order to avoid unnecessary work to answer the query (the output size is much smaller than the intermediate values). This is very similar to loop-order optimization, except that special care is taken to leverage sparsity of different join orders and thus cost estimates are maintained so that a quick search for "where to put the parentheses" can be done with dynamic programming. Similarly, **automated database design** is interested in which indexes to build and maintain (as well as many other options).

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.[19]

Many applications find that adding custom logic (i.e., a policy) for when to use what data structure can improve performance significantly.[20] More generally, however, data structures can have control parameters that allow them to interpolate between certain extremes. A great example is the adaptive replacement cache (ARC) (Megiddo and Modha, 2003), which is a 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) given a size constraint ($\mathsf{size} \leq C$). The two main features used by cache policies are *frequency* and *recency*. The ARC adapts to its (nonstationary) workload with a "learning rule." The way it works is by keeping around additional information about keys that *would have been* in the cache if it were 100% frequent (LFU cache), 100% recent (LRU cache), or something which is LFU x% of the time and LRU for the rest.[21] This additional tracking supports answering the following *counterfactual*: *What if the lru-lfu ratio had been x', instead of x?* At each time step, ARC adapts the ratio to be the best ratio according to recent memory, which is estimated quickly from the metadata kept.

Tuning solver heuristics: Hutter (2009) tuned the solver parameters of mixed integer

[18] Additionally, systems use problem instance features, such as input dimensions, to *dispatch* each instance to the best algorithm configuration for it. Of course, the dispatch policy also requires tuning. It also means that multiple versions of the code need to be generated and tuned for each instance category.

[19] Time-space complexities are not considered part of the ADT, but rather properties of a *specific* data structure. The ADT leaves many degrees of freedom. For example, an associative array can be implemented with a hash table, a linked list with linear search, a binary search tree, etc.

[20] In parallel applications, policies that govern locking and scheduling mechanisms are crucial for good performance. Adaptive policies for "smart locking," learned with RL, seem promising (Eastep et al., 2010).

[21] This means there is $2 \cdot C \cdot \mathsf{sizeof(key)}$ additional memory. We presuppose $\mathsf{sizeof(key)} \ll \mathsf{sizeof(value)}$ so the add memory is negligible in relation to the size of the cache.

programming solvers and satisfiability solvers with black-box optimization techniques according to specific workloads. The workload consists of problem instances which (more or less) arise naturally in a given setting, so the default setting might not have been tuned to exploit any regularity in the specific workload (e.g., maybe they are all easy SAT problems). Hutter (2009) won a number of competitions with his tuning approach, which demonstrated the effectiveness of tuning solvers to a specific workload.

Hutter (2009)'s work is especially exciting and relevant to auto-tuning Dyna, since Dyna too is an automatic solver (of fixed-point equations) with many opportunities for adaptive heuristics. Later in this document we will describe Dyna's major adaptivity categories. However, one important difference between our settings is that Hutter (2009) focused on the *off-line* setting, where the solver is treated like a *black-box* that is repeatedly called with different parameters. We're interested in *on-line* tuning because we don't expect users to sit and wait for several hours (or even days) while the Dyna solver runs their program hundreds or thousands of times. We're also interested in a more *glass-box* approach that leverages program structure.

Hyperparameter tuning: Nowadays machine learning approaches are riddled with decisions about fiddly little details, like optimization step-size schedules and update rules, SGD minibatch size, model architecture (e.g., number and size of hidden layers in neural networks). We now regard these choices as *hyperparameters* and have taken to optimizing them according to performance on additional held-out *development* data—possibly on cross-validated on multiple train-dev splits. Hyperparameter tuning is the new norm in machine learning as it has become important to ensure that approaches realize their full potential (Bergstra et al., 2013). There are many influential papers on the subject, which demonstrate its importance as well as developing new algorithms to do it efficiently e.g., Bergstra et al. (2013) and Snoek et al. (2012).

In general, hyperparameters may be smooth or not smooth. Discrete options, such as model architecture, are clearly not smooth. Real-valued parameters, such as regularization constants, step size schedules, can be tuned by gradient-based optimization (e.g., Maclaurin et al. (2015), Do et al. (2008)). Leveraging gradients allows us to scale tuning to thousands of hyperparameters.

End-to-end training: In Vieira and Eisner (2017), we showed how to do end-to-end training of pruning policies for a family of decoding algorithms for probabilistic models. This work uses reinforcement learning to tune a policy under an objective function that encourages accuracy while penalizing for runtime. We can think of pruning as a type of free-choice problem (section 3.6).

5 The research plan

5.1 The state of Dyna

In 2005, Dyna 1[22] was released (Eisner et al., 2005). It was pretty efficient, but only supported semiring-weighted programs and forward chaining for inference. Dyna 1 was truly ahead of its time supporting automatic differentiation, which is now a necessity in machine learning, 12 years later, with the rise of deep learning.

In 2013, Nathaniel Wesley Filardo and I built a prototype of Dyna 2.[23] Although this system only supported a subset of the Dyna 2 language, it was robust and usable enough for Jason Eisner to teach a summer course on NLP with assignments that made extensive use of our prototype. This system is not yet efficient because we use too many one-size-fits-all strategies. However, I believe that it is a good starting point for me to start experimenting with adaptive tuning.

I will extend our Dyna 2 prototype to support mixed-chaining and to generate more-specialized code and data structures. I will collaborate with Nathaniel Wesley Filardo and Matthew Francis-Landau to accomplish this goal. I will push to make this a tool that others will want to use.

5.2 Evaluation

Benchmarks My main plan is to take existing applications and implement them in Dyna. I will

[22] http://dyna.org

[23] http://github.com/nwf/dyna

recruit my lab mates and other future Dyna users to contribute. Here are some preliminary ideas:

- Tuning NLP pipelines on the existing problem workloads: constituency parsing, language modeling, machine translation.
- Using Dyna to support an *outer* algorithm: Algorithms, like MCMC or particle filtering, might interactively use Dyna as a subroutine or database. We can regard these algorithms as natural sources of queries and updates.
- Streaming data analytics
- Database benchmarks, which will include updates and queries.
- Shortest path on graphs, ideally *dynamic* graphs, e.g., route planning with traffic.

We will want to make sure that the benchmarks exercise Dyna query and update capabilities. Of course, running the program once to answer a single query is an important use case.

Evaluation: There are a few questions that are interesting to ask about the effectiveness of any technique that we might consider for adaptive tuning, which we will address empirically.

- How do our learned policies compete with existing methods, e.g., implemented by human experts or found by other algorithms?
- How important is dynamic adaptivity? Do we learn policies that can beat the best *fixed* policy from the same space?
- What is the overhead of training? Is adaptivity *really* worth the burn-in period?
- What is the overhead of the policy itself? Sometimes "thinking" about what to do adds too much overhead.

5.3 Timeline

- Extending Dyna 2 prototype (section 5.1)

  2-3 months of initial effort and background attention throughout

  Spring 2017

- Developing benchmarks, baselines and experimental infrastructure (Section 5.2)

  These things will come together naturally as needed. No specific deadline, will be background task.

- Query and update planning (Section 3.4) and Data structures selection (Section 3.5)

  Spring 2017

- Learning to memoize and index (Section 3.3)

  Summer 2017

- Learning to prioritize (Section 3.2)

  Fall 2017

- Free-choice operator (Section 3.6)

  Spring 2018

- Integration of all things, the grand unified model of everything, the thesis.

  Summer 2018

References

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

James Bergstra, Daniel Yamins, and David D. Cox. 2013. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *ICML*.

Shalabh Bhatnagar, Mohammad Ghavamzadeh, Mark Lee, and Richard S. Sutton. 2007. Incremental natural actor-critic algorithms. In *NIPS*.

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

Kai-Wei Chang, He He, Stéphane Ross, Hal Daumé III, and John Langford. 2016. A credit assignment compiler for joint prediction. In *NIPS*.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174.

John DeNero, Mohit Bansal, Adam Pauls, and Dan Klein. 2009. Efficient parsing for transducer grammars. In *NAACL*, pages 227–235.

Chuong B. Do, Chuan sheng Foo, and Andrew Y. Ng. 2008. Efficient multiple hyperparameter learning for log-linear models. In *NIPS*.

Aaron Dunlop, Nathan Bodenstab, and Brian Roark. 2010. Reducing the grammar constant: An analysis of CYK parsing efficiency. Technical report, CSLU-2010-02, OHSU.

Aaron Dunlop, Nathan Bodenstab, and Brian Roark. 2011. Efficient matrix-encoded grammars and low latency parallelization strategies for CYK. In *IWPT*.

Jonathan Eastep, David Wingate, Marco D. Santambrogio, and Anant Agarwal. 2010. Smartlocks: lock acquisition scheduling for self-aware synchronization. In *Proceedings of the international conference on Autonomic computing*.

Jason Eisner and Nathaniel W Filardo. 2011. Dyna: Extending datalog for modern AI. In *Datalog Reloaded*, pages 181–220. Springer.

Jason Eisner, Eric Goldlust, and Noah A. Smith. 2005. Compiling comp ling: Weighted dynamic programming and the Dyna language. In *HLT/EMNLP*, pages 281–290, Vancouver, October. Association for Computational Linguistics.

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

Nathaniel Wesley Filardo and Jason Eisner. 2012. A flexible solver for finite arithmetic circuits. In *ICLP*, pages 425–438.

Matteo Frigo and Steven G. Johnson. 1998. Fftw: An adaptive software architecture for the fft. In *ICASSP*, volume 3.

Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. 2016. Memory-efficient backpropagation through time. *CoRR*, abs/1606.03401.

Nikolaus Hansen. 2006. The cma evolution strategy: a comparing review. In *Towards a new evolutionary computation*.

Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. 2013. Bayesian optimization with censored response data. *CoRR*, abs/1310.1947.

Frank Hutter. 2009. *Automated Configuration of Algorithms for Solving Hard Computational Problems*. Ph.D. thesis, The University Of British Columbia (Vancouver).

Jack Kiefer and Jacob Wolfowitz. 1952. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, 23(3).

Scott Kirkpatrick, C Daniel Gelatt, Mario P Vecchi, et al. 1983. Optimization by simulated annealing. *science*, 220(4598).

Xiaoming Li, María Jesús Garzarán, and David Padua. 2004. A dynamically tuned sorting library. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*.

Adam Lopez. 2007. Hierarchical phrase-based translation with suffix arrays. In *EMNLP*, pages 976–985.

Dougal Maclaurin, David Duvenaud, and Ryan P. Adams. 2015. Gradient-based hyperparameter optimization through reversible learning. In *ICML*.

Nimrod Megiddo and Dharmendra S. Modha. 2003. Arc: A self-tuning, low overhead replacement cache. In *USENIX File & Storage Technologies Conference (FAST)*.

Anil Nelakanti, Cedric Archambeau, Julien Mairal, Francis Bach, and Guillaume Bouchard. 2013. Structured penalties for log-linear language models. In *EMNLP*.

John A. Nelder and Roger Mead. 1965. A simplex method for function minimization. *The Computer Journal*, 7(4).

Jervis Pinto. 2015. *Incorporating and Learning Behavior Constraints for Sequential Decision Making*. Ph.D. thesis, Oregon State University.

Sunita Saragawi. 2006. Efficient inference on sequence segmentation models. In *ICML*.

John Schulman, Nicolas Heess, Theophane Weber, and Pieter Abbeel. 2015. Gradient estimation using stochastic computation graphs. *CoRR*, abs/1506.05254.

Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. 2012. Practical Bayesian optimization of machine learning algorithms. In *NIPS*.

James C. Spall. 2003. *Introduction to Stochastic Search and Optimization: Estimation, Simulation, and Control*. Wiley Online Library.

Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. 1999. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*.

Tim Vieira and Jason Eisner. 2017. Learning to prune: Exploring the frontier of fast and accurate parsing. *TACL*.

Tim Vieira, Ryan Cotterell, and Jason Eisner. 2016. Speed-accuracy tradeoffs in tagging with variable-order crfs and structured sparsity. In *EMNLP*.

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

Ronald J. Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(23).

Geoffrey Zweig and Mukund Padmanabhan. 2000. Exact alpha-beta computation in logarithmic space with application to map word graph construction. In *ICSLP*.