Learning Speed-Accuracy Tradeoffs in Nondeterministic Inference Algorithms

Jason Eisner[1] **Hal Daumé III**[2]
[1] Center for Speech and Language Processing, Computer Science, Johns Hopkins University
[2] Computational Linguistics and Information Processing, Computer Science, University of Maryland
jason@cs.jhu.edu, me@hal3.name

1 Problem Definition

Statistical learning has led to great advances in building models that achieve high accuracy. However, test-time inference in these models can be slow, for example in structured prediction problems. This is frequently addressed by using test-time heuristics to guide and prune the search for a good structured output. In this high-level paper, we ask: Could we explicitly train such heuristics to trade off accuracy and efficiency? And how does this relate to existing learning problems?

We consider problems where there is a problem-specific notion of accuracy, and the function we wish to maximize is $(\text{accuracy} - \lambda \times \text{runtime})$, where λ is a user-defined parameter specifying the desired tradeoff.

More specifically, we focus our discussion on the problem of natural language parsing under a weighted context-free grammar. A parsing algorithm is typically nondeterministic, in that it explores many paths in parallel: it builds many constituents (sub-parses) that will turn out *not* to be subtrees of the final, highest-weighted parse tree. We assume a bottom-up architecture similar to a forward-chaining theorem prover (Kay, 1986; Eisner et al., 2005). Whenever a new constituent is built, it is inserted into a priority queue, the "agenda." At each step of the algorithm, a prioritization heuristic (Caraballo & Charniak, 1998) pops the highest-priority constituent from the agenda. This is combined with previously popped constituents to yield larger constituents that are then pushed back on the agenda. Our goal is to *learn* a heuristic that pursues promising constituents early on, and thus tends to lead quickly to a high-accuracy parse. We can tolerate a little search error, so we are not restricted to the admissible heuristics of A* search (Klein & Manning, 2003).

This is fundamentally a *delayed reward* problem: one does not know until parsing has finished how fast or accurate the parser was. We therefore cast learning as an off-line reinforcement learning problem. The prioritization heuristic is parameterized by θ. At training time, the reinforcement learning algorithm tries to find a parameter vector θ that optimizes a given reward, $R = (\text{accuracy} - \lambda \times \text{runtime})$, over the input distribution from which training data are drawn.

A well-studied setting for reinforcement learning is the Markov decision process (MDP), a formalization of a memoryless search process. An MDP consists of a (typically finite) *state space* $\mathcal{S}$, an *action space* $\mathcal{A}$, a (possibly stochastic) *transition function* T, and a (possibly stochastic) *reward function* R. An agent observes the current state $s \in S$ and chooses an action $a \in \mathcal{A}$. The environment responds by moving the agent to another state $s' \in \mathcal{S}$, which is sampled from the MDP's transition distribution $T(s' \mid s, a)$, and by giving the agent a reward r, which is sampled from the MDP's reward distribution $R(r \mid s, a, s')$. The agent's goal is to maximize its total reward over time. An agent's *policy* π describes how the agent chooses an action based on its current state.

For our parsing example, the state consists of the input sentence, the agenda, and the parse chart (the set of previously popped constituents). So the state space is astronomically large. The agent controls which item (constituent) to pop next from the agenda. Thus, in any state, the possible actions correspond to items currently on the agenda. The agent's policy π is our prioritization heuristic, parameterized by θ, which selects an item y and pops it. The environment responds *deterministically* by adding y to the chart, combining y as licensed by the grammar with various adjacent items z that are already in the chart, and placing each resulting new item x back on the agenda (with duplicates removed in favor of the higher weight one). Parsing ends when π chooses to pop the special action HALT. Then the highest-weight complete parse to date is returned, and the environment rewards the agent for the accuracy and speed of that parse.

We consider deterministic policies that always pop the highest-priority available action. Thus, learning a policy corresponds to learning a priority function. We define the priority of action a in state s as the dot

product of a feature vector $\phi(s, a)$ with the weight vector θ. Note that features can depend not just on the action (the item being popped) but the state (the input sentence, and the chart and agenda of constituents that have been built so far). *Static features* are those that depend only on the action and the original parser input; *dynamic features* are those that change along with the state. This terminology derives from the fact that even after an item y is pushed onto the agenda, the chart may change and therefore the corresponding values for some of its dynamic features may change. In particular, the item y might be encouraged/discouraged as the parser discovers other constituents or configurations of constituents that are compatible/incompatible with y.

2 Relationship to Typical Reinforcement Learning Problems

There are several key ways in which this problem differs from typical reinforcement learning settings. Some of these make the learning problem harder, but others make it potentially easier.

Note first that our learning is *episodic*. Each run is a separate and finite "episode," whose initial state is determined by an input sentence to be parsed. We assume the *offline* learning setting, with separate phases for training (exploration) and testing (exploitation). We will be evaluated only on the reward earned during testing episodes; so we are not penalized for low-reward episodes at training time.

Challenges. The primary difficulties relate to the scale of the problem. The astronomical size of the state space makes it hard to do value estimation or planning.[1] The choice among hundreds of actions at each state makes it hard to identify good actions. And a typical episode consists of thousands of actions, making it difficult to say which of these actions deserve credit for the eventual reward.

In addition, we will eventually wish to search through a high-dimensional policy space, particularly when we use dynamic features. This is because the usefulness of a parse action may be sensitive to detailed configurations of the parse chart and input, which motivates generating large sets of potentially useful features. Unfortunately, the expected reward as a function of the policy parameters can be highly non-convex. For example, there exist parsing strategies, such as coarse-to-fine strategies, in which extra items are built solely to trigger dynamic features that affect the priority of the traditional items (Felzenszwalb & McAllester, 2007). But discovering such coordinated strategies by exploration is difficult, because producing the extra items is a waste of time until one has also learned to consume them.

Opportunities. Fortunately, our setting also has attractive properties that enable some novel approaches to learning. (These properties are not unique to parsing, but are shared with many other AI computations, including machine translation, image parsing, and constraint propagation.)

First, our goal is to learn how to act within a simple idealized model of computation. The environment responds deterministically to our actions (even if we decide to explore the search space by choosing some actions randomly). Thus, we can run controlled experiments where we change part of the policy, or specific actions within an episode, to see what happens.

Even better, we have perfect knowledge of the environment. Thus, as discussed below, we may be able to reason analytically about the results of certain such changes. This may be much faster than testing the changes by running new episodes, particularly if we wish to reason about large batches of episodes (e.g., the average reward of a stochastic policy). Because we know the causal structure of the environment, we may also be able to reason backward to identify changes that could bring about a desired improvement. Even where *exact* answers would require simulation of new episodes, we may still be able to do effective *approximate* reasoning using a simplified model. This is analogous to using variational inference in place of MCMC simulation.

Second, our environment is mainly an arena for monotonic reasoning. A parser is a kind of theorem prover that gradually discovers all facts that are provable from the input—e.g., the fact that a certain substring of the input *could* be analyzed as a Noun Phrase constituent. As the parser runs, the set of discovered facts (i.e., popped items) only becomes larger. This setting has three striking properties:

- Facts are renewable resources. Once an item z has been built and added to the *chart*, it remains available forever, for use as a subconstituent in any number of larger constituents.
- Once an item y has been pushed onto the *agenda*, the action of popping it remains available to the agent indefinitely, until that action is taken.
- If y and z are provable and the set of provable items is finite, then the parser will eventually prove both y and z, if it runs for long enough.[2]

[1] Indeed, there are very many reasonable plans for achieving a high reward, since there are many routes to the same parse. However, it is not known which plans can be achieved within our policy space. The fastest plans pop only correct constituents, with no wasted work, but presumably no *policy* exists that is so clarivoyant.

[2] This situation no longer holds if we allow actions that permanently *prune* parts of the search space, rather than simply *postponing* exploration of these parts through the prioritization heuristic. Heuristic pruning is commonly used in parsers.

Together, these properties mean that the environment is quite forgiving about the *ordering* of actions (which is what our policy really controls). If the agent does not choose to pop y now and combine it with z to obtain x, it will still be able to do so later. Moreover, the resulting delay in producing x will not reduce the opportunities eventually available to x. Hence small changes to the policy, which reorder items slightly on the agenda, tend to have small and predictable effects on the trajectory through state space. This is different from many reinforcement learning settings, where a single action may radically change the state, the available next actions, and the eventual reward.

This situation is somewhat complicated by dynamic features. Although an item, once proved, remains around forever as an immutable fact on the agenda or chart, its *priority* on the agenda is not immutable but may fluctuate as the item's dynamic features change. For example, it may be important for the agent to discover evidence that the item should have low priority *before* it unnecessarily pops the item and triggers a cascade of useless work. Still, the agent can largely recover from such poor decisions.

Third, we can sometimes exploit the nature of our reward function $(\text{accuracy} - \lambda \times \text{runtime})$. For a given input, it is often tractable to find a high-*accuracy* trajectory through state space, for example by running the agenda-based computation for long enough with a simple initial policy.[3] Policy learning can then focus on increasing the priority of the items that already appear in a desirable trajectory or a set of desirable trajectories, so that they—and the HALT action—are popped earlier. Here the accuracy problem ($\lambda = 0$) has already been solved by previous work; we are merely trying to increase speed ($\lambda > 0$).

3 New Reinforcement Learning Strategies

For several reasons, we find it unlikely that classic reinforcement learning algorithms like policy gradient (Williams, 1992; Sutton et al., 2000) or conservative policy iteration (Kakade & Langford, 2002) will find reasonable policies in a tolerable amount of time. The estimates of the policy gradient have high variance. Even the true policy gradient is not all that useful: it tends to be quite flat, because *small* changes to the policy will at best reorder actions locally in a way that does not affect overall reward. Indeed, when we ran a small policy gradient experiment with just 20 features, the policies it found were significantly worse than a simple hand-coded policy that used only 2 features.

We would instead like to determine how to adjust the policy by reasoning about causality in our environment. We may reason that popping item y *much* earlier would enable or encourage another desirable item to be popped early. Similarly, we may reason that *greatly* lowering the priority of a apparently unhelpful item, to below the priority of STOP, would prevent that item from popping before STOP and triggering additional useless work.

Intuitively, for a given input sentence w, such reasoning should allow us to adjust our policy in a direction that would tend to eventually improve the speed and accuracy of parsing w. Such "what if" analysis is imaginable because of the special properties of our setting, as discussed in section 2.

Our outer loop is typical of policy search:[4] At time t, our policy is determined by parameters θ_t. We consider the trajectory $\bar{\tau} = \tau(w_t, \theta_t)$ that this current policy would take on some randomly chosen input sentence w_t, and choose θ_{t+1} so that higher-reward trajectories for w_t are *actually* chosen or *come closer* to being chosen.

One version of this searches for a few specific, complete trajectories that are "near $\bar{\tau}$" but have higher reward. The parameters θ can then be updated toward these trajectories. E.g., in our setting, we can efficiently identify individual actions whose priority it would be profitable to change (keeping the rest of the policy the same).[5]

An alternative type of adjustment is a kind of "surrogate policy gradient." How would our reward on w_t change with θ? Consider the function $R_t(\theta)$ that gives the reward of trajectory $\tau(w_t, \theta)$. Unfortunately, this function is piecewise constant.[6] So based on the current trajectory $\bar{\tau} = \tau(w_t, \theta_t)$, we construct some highly smoothed, differentiable surrogate function $\tilde{R}_t(\theta)$ that approximates $R_t(\theta)$, at least for θ in the vicinity of the current parameters θ_t. We then choose θ_{t+1} by following the gradient of $\tilde{R}_t(\theta)$. We propose to find this gradient analytically, not by sampling as in ordinary policy gradient. Below are two versions of this idea.

[3] Agenda-based parsing will eventually find the highest-weighted parse. For basic approaches to parsing (e.g., weighted context-free grammars of natural language), the runtime of doing this is a manageable polynomial in the length of the sentence, and state-of-the-art models (grammars) are good enough that the highest-weighted parse is fairly accurate with respect to the gold standard. All this is somewhat less true for the formally similar problem of machine translation.

[4] Loss-sensitive learners for structured prediction also have this flavor (Crammer et al., 2006; McAllester et al., 2010).

[5] Candidate actions can be identified as in section 3.2 below. We use change propagation to determine how moving a candidate action's time within τ would affect the reward: a few intervening actions would also move in time because they would be pushed earlier/later or would pop with a different priority. While these changes may cascade somewhat, computing them should still be faster than recomputing the trajectory from scratch.

[6] Small changes to θ preserve the trajectory, except when they make a new action become the highest-priority action.

3.1 Variational Surrogate

$R_t(\theta)$ gives the reward of the single trajectory $\tau(w_t, \theta)$. To smooth this, we would like $\tilde{R}_t(\theta)$ to measure the *expected* reward of a bundle of trajectories in a neighborhood around $\tau(w_t, \theta)$. Such a bundle can be defined by adding noise to the policy defined by θ. We can then take our gradient step $\theta_{t+1} = \theta_t + \epsilon \nabla \tilde{R}_t(\theta_t)$.

So far, this is just policy gradient. However, policy gradient would usually sample trajectories from the "current" bundle (around $\bar{\tau} = \tau(w_t, \theta_t)$) to estimate their expected reward $\tilde{R}_t(\theta_t)$ and its gradient. Sampling has 0 bias, but very high variance in our setting. We wish to improve the bias-variance tradeoff. Our setting lends itself well to variational approximation, for reasons discussed in section 2: within a bundle of trajectories, actions at different timesteps are fairly independent of one another, as are the various features of the state.

Thus, we compute a variational approximation to the joint distribution over the current bundle's trajectories and their rewards. Using this approximation, it is possible to estimate the current bundle's expected reward $\tilde{R}_t(\theta_t)$. As for its gradient, variational approximation is not a closed-form computation, but we can still use algorithmic differentiation to determine how our estimated reward would have been affected by changes to θ.

The use of a distribution over trajectories—the "bundle"—is what makes $\tilde{R}_t$ differentiable. To define it, we recommend adding a unique feature for each item y. A trajectory is sampled by running the policy *deterministically* after sampling these features' weights $\theta_y \sim \mathcal{N}(0, \sigma^2)$. When $\theta_y < 0$, y tends to pop later. This scheme finds a more diverse trajectory bundle than Boltzmann exploration.[7] We decrease σ^2 over time.

3.2 Priority-Based Surrogate

We consider another design for the approximate reward model $\tilde{R}_t(\theta)$. Intuitively, it is easy to see where the current trajectory $\bar{\tau}$ would benefit from popping certain constituents much sooner or later. To make x pop sooner, we must raise its priority and/or the priorities of the subconstituents from which it was built. We will raise them slightly taking a step along the gradient of $\tilde{R}_t$. The idea is that the true R_t is locally flat, but the gradient of $\tilde{R}_t$ should point toward faraway points that would actually displace x enough to affect the reward.

We can formalize the idea as follows. We do not estimate the reward of each possible $\theta \neq \theta_t$ by actually revising the trajectory $\bar{\tau}$ (or bundle of trajectories around $\bar{\tau}$). Instead, from θ we construct a revised estimate u_x of each item x's "effective priority" (defined below), such that items with larger u_x tend to pop sooner. Our reward estimate $\tilde{R}_t$ is defined to be a smooth function of the u_x values. Thus, following the gradient means changing θ in a way that changes the u_x values so as to increase this smooth $\tilde{R}_t$ fastest.

The basic insight is that reward = accuracy $- \lambda \times$ runtime is determined by *which items x pop before the* HALT *action*, i.e., $u_x > u_{\text{HALT}}$. Popping x before HALT hurts runtime (since the parser then tries to build new items from x). But occasionally it also helps accuracy by changing the parser output (= the top-scoring complete parse that is popped before HALT). Given the current trajectory $\bar{\tau}$ (obtained by running the policy well past HALT, with parameters θ_t), we can easily compute these direct effects: the change Δ_x to the current reward $\bar{R}$ if x by itself moved to the other side of HALT. We now define $\tilde{R}_t = \bar{R} + \sum_x \frac{\Delta_x}{2} - \frac{\Delta_x}{2} \cdot \frac{\tanh((u_x - u_{\text{HALT}})/T)}{\tanh((\bar{u}_x - \bar{u}_{\text{HALT}})/T)}$ where $\bar{u}_x$ and $\bar{u}_{\text{HALT}}$ are the effective priorities of x and HALT in the current trajectory $\bar{\tau}$. This estimates that the reward of the current trajectory would increase by Δ_x if x were to move all the way to the other side of HALT (preserving the distance between u_x and u_{HALT} but reversing its sign). It interpolates by supposing that as x moved in that direction, $\tilde{R}$ would rise gradually along a sigmoidal curve, changing fastest as u_x crossed u_{HALT}. The steepness of the sigmoid is controlled by temperature T. A high temperature gives a nearly linear function, so that $\frac{\partial \tilde{R}}{\partial u_x} \approx \frac{\Delta_x}{2(\bar{u}_{\text{HALT}} - \bar{u}_x)}$. A low temperature gives a locally flat function that better approximates the true reward function; in this case, we do not try as hard to move u_x toward or away from u_{HALT} if it is currently far away from it. We can reduce temperature as learning proceeds.

We use "effective priorities" u_x rather than actual priorities v_x to deal with the fact that items do not always pop in priority order. Even a high-priority item may pop late if it is built from low-priority items. To pop x early, we certainly need its actual priority v_x to be high, but we also need to have *pushed* x early. Suppose the grammar says that x gets pushed whenever we pop and combine either (y **and** z) **or** (y' **and** z'). Thus, we define $u_x = \min(v_x, \max(\min(u_y, u_z), \min(u_{y'}, u_{z'})))$. Here min corresponds to **and** and max corresponds to **or**. If $\tilde{R}_t(\theta)$ increases with u_x, this definition shows that to increase it, one may have to increase not only x's own priority v_x, but also the priorities of a sufficient set of ancestors of x that must pop before HALT in order for x to do so. We smooth the u_x function by using softmin and softmax in place of min and max.

[7] Suppose that y is the highest-priority action at time t. Boltzmann exploration may sometimes pop a lower-scoring acton y' at time t instead. However, under Boltzmann exploration, y' will retain a high priority and a high probability of popping at time $t+1$ or $t+2$, so it is unlikely that y' will be delayed long enough to change the reward.

The u values depend on v values. We define v_x in turn via the usual linear function $\theta \cdot \phi(s, x)$, where $\phi(s, x)$ extracts features of x in state s. But the state s varies with θ. For example, a certain dynamic feature $\phi_k(s, x)$ may fire only if s reflects that y has previously popped into the chart. Our approximation $\tilde{R}$ is based on u values rather than states, so we say that y pops (and the feature fires) in time to affect v_x iff $u_y > u_x$. Again we interpolate sigmoidally, gradually sliding the feature value from 0 to 1 or from 1 to 0 as u_y passes u_x.

All this means that the u and v values are defined by a recurrent system of equations. In practice, we estimate them at $\theta = \theta_t$ by a few rounds of iterative update, and use these u values to compute $\tilde{R}_t(\theta_t) = \bar{R}$. By applying algorithmic differentiation as before, we can compute the gradient of this result with respect to θ.

4 Conclusions

We have described a new setting for learning, namely the speed/accuracy tradeoff in non-deterministic inference algorithms like agenda-based parsing. Because of the enormous search space in these problems, off-the-shelf reinforcement learning algorithms fail miserably. We have identified three unique characteristics of our problem, and suggest mechanisms to exploit these to obtain more feasible learning algorithms.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 0964681. We would like to thank our students Jiarong Jiang, Adam Teichert, Tim Vieira, and He He for useful discussions and preliminary experiments.

References

Bagnell, J. A., Kakade, S., Ng, A., & Schneider, J. (2003). Policy search by dynamic programming. *Advances in Neural Information Processing Systems (NIPS)*.

Caraballo, S. A., & Charniak, E. (1998). New figures of merit for best-first probabilistic chart parsing. *Computational Linguistics*, *24*, 275–298.

Crammer, K., Dekel, O., Keshet, J., Shalev-Shwartz, S., & Singer, Y. (2006). Online passive-aggressive algorithms. *Journal of Machine Learning Research (JMLR)*, *7*, 551–585.

Eisner, J., Goldlust, E., & Smith, N. A. (2005). Compiling comp ling: Weighted dynamic programming and the Dyna language. *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing* (pp. 281–290). Vancouver: Association for Computational Linguistics.

Felzenszwalb, P. F., & McAllester, D. (2007). The generalized A* architecture. *Journal of Artificial Intelligence Research*, *29*, 153–190.

Kakade, S., & Langford, J. (2002). Approximately optimal approximate reinforcement learning. *Proceedings of the International Conference on Machine Learning (ICML)*.

Kay, M. (1986). Algorithm schemata and data structures in syntactic processing. In B. J. Grosz, K. Sparck Jones and B. L. Webber (Eds.), *Readings in natural language processing*, 35–70. Los Altos, CA: Kaufmann. First published in 1980 as Xerox PARC Technical Report CSL-80-12 and in the Proceedings of the Nobel Symposium on Text Processing, Gothenburg.

Klein, D., & Manning, C. (2003). A* parsing: Fast exact Viterbi parse selection. *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics and Human Language Technology (NAACL/HLT)*.

McAllester, D., Hazan, T., & Keshet, J. (2010). Direct loss minimization for structured prediction. *Advances in Neural Information Processing Systems (NIPS)*.

Sutton, R. S., McAllester, D., Singh, S., & Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. *IN ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 12* (pp. 1057–1063). MIT Press.

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