Toward a New Approach to Model-based Reinforcement Learning

Richard S. Sutton

May 6, 2020

Perhaps the biggest open question in reinforcement learning (RL) is that of how the agent represents its knowledge of the environment and relates its knowledge to the data stream of experience. In principle, everything to be known about the environment is a fact about the agent's data stream and thus could be learned or verified from the data stream without external supervision. The regularities in the stream are complex and subtle, but modern computers are large and fast, and the machines of the future will be ever more so. RL researchers should be exploring ambitious agents that learn complex approximations to the data stream's regularities and then use that knowledge by computationally intensive planning processes to make decisions. We need algorithms for finding the knowledge efficiently and for representing it in a way that is expressive, efficiently learnable, and suitable for planning. Of course, this is key challenge not just for RL, but for artificial intelligence (AI) in general. RL just gives us a particular clear setting in which to explore the challenge. In AI, the challenge of interrelating learning, representing, and planning is sometimes referred to as the problem of commonsense knowledge and commonsense reasoning.

Modern model-based RL is directly addressing these challenges. In model-based RL, world knowledge is represented as a *model of the environment*. The environment model comprises a *state part* and a *dynamics part*, both of which should in general be constructed by the agent. We formulate the state part as the learning of a *state update function*, which deterministically generates the new state from the previous state and the latest elements of the data stream. The state update function can be thought of as a recurrent artificial neural network. The dynamics part of the environment model makes action-conditional predictions about state transitions; we formulate it as a set of *general value functions* (GVFs). A key part of the data stream are the *actions* decided on and taken by the agent at each step. A stored *policy* maps the state produced by the state update function at each time to the action to be taken, or to a probability distribution thereof. The policy is determined by learning processes using the data available at each time step and, in the case of a model-based agent, by planning processes using the environmental model. The distinguishing feature of a *model-based* RL agent is that much of its policy learning is mediated by the environment model; the model is learned, and then planning with the model influences the stored policy.

We seek to leverage the data as much as possible, with the least amount possible being built in by people. This means no built-in features, options, or questions other

than the base questions about the value function and the optimal policy (or other seed things to give an ultimate grounding to grow around). All these things would be searched for and learned. We presume no objects, no other agents—just a data stream, or signals and weights. No self. We can build in generic models and state-update functions, policy and value questions. But we accept the discipline that nothing else is built in. All else is learned or discovered. All exploration, all behavior. All choices of state, all ways to behave above the primitives. We can build-in generic search procedures, generic discovery algorithms, and of course generic learning algorithms. There should be no hyperparameters.

There are at least three interrelated issues involved in model-based RL: 1) discovering and learning state, 2) discovering and learning a model of the dynamics (state to state), and 3) planning effectively and efficiently with the model. All three issues are strongly interlinked. All of them are needed in order to properly/fully assess each component: to assess state one needs to see what one can predict from it; to assess the state-to-state dynamics, one needs the state; and to assess planning one needs the model, and one needs a changing world.

In the following sections, we discuss notions of state and state features (Section 1), planning by value iteration (Section 2), models of the environment's dynamics, particularly expectation models (Section 3), subproblems based on state features (Section 5.2), and finally the cycle of discovery which pulls all the parts together (Section 5). A general learning algorithm used by many of the parts is presented in Section 4.

1 State

An essential part of any agent architecture, even a model-free one, is the agent's representation of where it is at the current time—its computation of something that it will use as an approximation to the state in a classic Markov decision process (MDP). The state is input to the agent's policy and to the agent's state-value function, and it is both input and output of the environmental model in a model-based agent. Maintaining the agent's sense of state is analogous to the process of perception in psychology. Learning how to maintain state corresponds to the representation learning. Let us examine various notions of state in detail.

1.1 Agent state

The most important notion of state in reinforcement learning is what we term "agent state". The *agent state* at time t is a compact summary of the agent's trajectory up to t (past actions and observations, including rewards) that is good for predicting and controlling the future of the trajectory (that is, for use by predictions and policies). One major activity for an intelligent agent, then, is to construct its agent state. The agent will learn how to do this through experience while simultaneously learning to get reward and to estimate value, and perhaps making a host of other predictions including those making up its model of the environment.

Note that the agent state is different from the environment's state, to which it

is only distantly related. The state of the environment is all the things used by the *environment* to compute itself (and the observations and rewards, based on the agent's actions). The state of the environment is full of things that are far from any observations, such as what is going on in remote parts of the world, or in distant galaxies. The state of the environment includes latent states that may have a physical reality but are not immediately visible, like x and y coordinates or the underlying states of a Partially Observable MDP. In a POMDP, the agent state is the distribution of possible environment states. The number of agent states could in principle be either smaller or larger than the number of environment states, but in typical real-world cases it would be vaster smaller, in accord with the aperture principle (Sutton et al., 2018, Section 2).

The agent state should not be confuse with the state of the agent (as the most straightforward interpretation of the term "agent state" might suggest). The state of the agent includes all the weights and other parameters and hyper-parameters that the agent might adapt over time. The agent state, on the other hand, is only the agent's summary of the history of the current interaction. For example, in an episodic problem, in which the environment is reset at the start of each episode, the agent state would also be reset at this time. Any learning by the agent in past episodes is not part of the agent state, but of course it is part of the complete state of the agent. Because of this, the term "agent state" is perhaps not the best. Perhaps "agent's history summary" would be better.

We focus on the agent state because it is the input to the policy, to value functions and predictions, and to the model of the environment. It is a central, key component of the agent architecture. It is only right that we should have a special, compact name for it. The agent state plays the role that, in the fully observable case, would be played by the MDP state, so it seems natural, most of the time, to refer to it simply as *the* state. It is close to the MDP state, and there will be nothing closer to it available to the agent. The agent state is where the agent has gotten to, ideally in an independence-of-path kind of way. It is where the agent is now, its current situation, as best as the agent knows it or has remembered it. It is the agent's perception of where it is now.

Finally, note that the agent state is explicitly meant to be an imperfect approximation. It is whatever the agent has, not what it should have. The agent state is distinct from the ideal notion of state that is sometimes called the *information state*. The space of possible histories can be partitioned into subsets, all elements of which are equivalent in their probabilities over future trajectories conditional on action. After merging all equivalent subsets, then the index of the subset containing the current history is the information state. This is the information-theoretic ideal for the agent state.

A state being an information state is a stronger condition than its being merely *Markov*. Markov generally requires that all informationally significant distinctions are retained in the agent state, but allows for distinctions that are not informationally significant, whereas information state means that, in addition, no distinctions are retained that are not needed. The information state is thus potentially more compact. There is more useful discussion and definitions of the Markov property in Chapter 17 of the RL text and in the slides of Andre Barreto's recent talk.

1.2 State Update

The agent state is a summary of the history, so conceptually it is a function of history. However, histories can become quite long. For computational efficieny, we prefer to compute the agent state S_t as an update from the previous state S_{t-1} together with the most recent observation O_t and action A_{t-1}. We denote the update function by u:

$$S_t \doteq u(S_{t-1}, O_t, A_{t-1}). \tag{1}$$

If the environmental (MDP) state is fully observable, then an agent might use it as agent state (Markov, but not necessarily an information state), yielding tabular methods. But if the state space is large, then the MDP state is often reduced to a feature vector, as in Part II of the RL text, and then that vector is used as agent state. In general, we would like to move toward a feature-vector view of agent state and away from the information and Markov ideas that the agent state is an index into a discrete partitioning of histories.

1.3 Feature-based State

Because the agent state is an approximation, we would like to be able to talk about better and worse agent states and, in the feature-vector view, about better and worse (agent) state features. We want to learn the agent state (by learning the state-update function u) and this is much easier done feature-by-feature rather that as an overall partitioning. The general idea is that an agent-state feature is better (worse) to the extent that it is useful (useless) for predicting and controlling the future trajectory. There are a number of aspects to this. There are several uses of the agent state, for example, by the policy, by predictions, and by the model of the environment. More fundamentally, we can distinguish between useful for performance and useful for learning. An agent-state feature can be good because it is currently used by the policy/predictions/model (performance), or it can be good because, if there is error, then its outgoing weights should be adjusted by the learning process (learning). These are two independent reasons and either of them may apply independently of the other. All four cases can occur.

Now let us consider the multiple uses, for performance or learning, that a state feature might serve. It can be used by the computations of any of the policies or predictions, or by the model. State features can also be used as prediction targets, for example in the cumulants of GVFs. In particular, they would be used explicitly as targets in making up the model. In all these cases, state features inherit the utility of the thing they help. That is, usefulness, either for performance or for learning, starts with some designated things of intrinsic usefulness by fiat, such as rewards and reflexes, and then propagates backward (not like back-propagation, but in a slow way) from there. We might call this *utility-propagation*. Utility propagation may also be grounded in learning itself, as in curiosity, or "learning feels good."

The informational and Markov ideals for a state are about its information content. Because of this, these ideals are more absolute and binary than is desirable. That is, they make for an entirely binary criterion of usefulness: the state's distinctions

either do or do not make a difference for some future. For our purposes, steeped as they are in approximations, we prefer a more continuous measure of usefulness. We care how much of a difference a feature makes, for which predictions, and for which futures. We need to abandon information-theoretic differences and embrace continuous approximations of usefulness. This opens up possibilities. We could, for example, seek features that can contribute linearly to accurate prediction or data-efficient learning.

1.4 Feature Discovery

Let us close this section by summarizing the view of agent state as comprising many features. Each feature will have one or more measures of its utility, perhaps separately for performance and for learning, and perhaps separately for its many uses; these will represent in some sense the degree to which this feature should be considered a state feature (and other features that do not yet meet this test, but are nevertheless being tried out, are *candidate* state features). The measures also guide the construction of the state-update function u. The main remaining task is to detail how this should be done.

The simplest and clearest strategy for constructing the state-update function (if not necessarily the most data efficient) is just generate and test, as in Mahmood's "online representation search" (Mahmood 2017, Chapters 11 and 12). This can also be combined with gradient descent as illustrated in Mahmood's work. Except really it should be in a context of reinforcement learning with sequential data and thus remembering of past signals (rather than, as in Mahmood's work, in the context of supervised learning and the GEOFF task). The network should also be a recurrent network as envisaged in Rich's talk on representation learning. There could also be a special kind of gradient descent whose goal is to produce features that can generalize well (which maybe implies linearly), as suggested by work on cross-propagation (Veeriah, Zhang and Sutton 2017). These existing ideas are enough to get started and produce systems that we fully understand and can follow along with. So we turn next to the other intertwined parts of the learning/planning agent architecture.

The agent state is input to all the other things to be learned. The most important of these are the policy and value function, and close behind are other predictions designated by the designer as important and probably linked to responses as in Pavlovian control (Modayil and Sutton 2014). All of these can be learned in the usual way and provide ultimate grounding to the measures of utility of the state features. Exactly how they determine the measures is yet to be worked out, but presumably each of these (policy, values, predictions) has an intrinsic degree of utility, and then features that enable their computation inherit that utility to the extent that they are important in computing them. For example, one feature could be important by having a high non-redundant weight and by being non-zero frequently. A slow utility-propagation process could compute this. Once a feature obtains utility it can then in turn propagate utility to features that it uses.

We now present the essential aspects of a complete model-based reinforcement learning agent, including discovery of state features and a feature- and option-based dynamics model. The discovery processes are centered around the state features.

In notation, we continue to dedicate the superscript position to the feature index. Thus we have feature i, with value x_t^i at time t, for $i = 1, \ldots, n$. These values can be arranged in an n-vector $\mathbf{x}_t$, which we can assume without loss of generality is a function of some otherwise unobservable underlying state ($\mathbf{x}_t \doteq \mathbf{x}(S_t)$).

Each feature will maintain a variety of scalar statistics, including its age—a measure of how long it has existed in more or less its current form—and various things about its distribution of values—maybe a mean, a variance, a skew? Maybe how much the other signals are active when this one is active. The notion of "active" is that the signals are roughly binary, 1s and 0s, active and inactive, present and absent. If it was pure binary, then its distribution would be wholly characterized by its activity probability/frequency. If it was only binary-ish, meaning values intermediate between 1 and 0 occur with some frequency, then the degree of binarity could be recorded as a statistic. And statistics could be kept for scalar measures of the activity of the other features conditional on this feature's activity.

Each feature potentially plays a role as input to many other things, with weights and step sizes. What are some of these things? State update. Models. Value functions. Misc other predictions? Each feature should have a scalar measure of valuableness that says how important it is to the system, and thus how critical it is to preserve the feature in more or less its current form. A feature's valuableness can only be assessed over an extended period of time, and thus should change very slowly. A limited number of features that have been valuable for a long enough time are *tenured*, meaning that they are allocated additional resources and become even less likely to change. The non-tenured features are termed candidates. Once tenured, it is rare for a feature to be demoted to a candidate again, even if it becomes much less valuable than existing candidates.

How valuableness should be measured is not completely clear, but it will depend on the extent to which the feature is used as input to the various things identified briefly in the previous paragraph and some of which are detailed further below. In general, there may be two kinds of valuableness, one corresponding to the current operation of the system (as evidenced by non-zero outgoing weights) and one corresponding to the way the system changes by learning (as evidenced by large step sizes on outgoing weights). The primary determinant of tenure as used here should be the magnitude of outgoing weights.

2 Planning

The dynamics part of an environment model takes a state and an option and projects to the agent-state-feature vector at termination of the option (when started in the state) plus the expected cumulative reward along the way. This process is called *projection*. It is the operation at the heart of planning. For a particular option, the model is an organized set of predictions, as explained in Chapter 17 of the RL text: If there are n state features, then there are n+1 GVFs, n to predict the state features at the end of the option (cumulant is $1-\gamma_t$ times the state feature) and one more to predict the cumulative reward during the option (cumulant is reward).

Planning can be conceived of in several different ways, but one of the simplest and

clearest is as approximate value iteration, which proceeds in sweeps over the entire state space, updating the approximate value of each state, $v_{\mathbf{w}}(s)$, based on the best of the model's one-step projections and the estimated value at the resulting state:

$$v_{\mathbf{w}}(s) \leftarrow \max_a \left[\hat{r}(s,a) + \sum_{s'} \hat{p}(s'|s,a) v_{\mathbf{w}}(s') \right], \tag{2}$$

where $\hat{r}$ and $\hat{p}$ are the reward and next-state parts, respectively, of the model of the action a. In an accurate, tabular, one-step model, $\hat{r}$ is the one-step expected reward function and $\hat{p}$ is γ times the state-transition probability for s, a. Classically the value function is a table, in which case the arrow is interpreted as assignment, but we are of course more concerned with approximate methods, in which case the arrow indicates an update to the parameter vector $\mathbf{w}$ that moves the approximate value toward the quantity on the right-hand side. In the classical case, the maximum in (2) is over the actions available in state s. However, without any other changes we can consider a to be an *option* and the maximum to be over the set of options available in s, using standard option models for $\hat{r}$ and $\hat{p}$ (Sutton, Precup & Singh 1999, Sutton & Barto 2018). Henceforth we assume this, without limitation, as actions are special cases of options.

The quantity in brackets in (2) occurs frequently when discussing planning. Let us call it the *backed-up value* of state s and option o, denoted $b(s,o)$:

$$b(s,o,v) \doteq \hat{r}(s,o) + \sum_{s'} \hat{p}(s'|s,o) v(s'). \tag{3}$$

Notice that this definition assumes an implicitly given model, $\hat{r}$ and $\hat{p}$. Using this notation, planning by value iteration is done by repeated updates (2) that can now be written simply

$$v_{\mathbf{w}}(s) \leftarrow \max_o b(s,o,v_{\mathbf{w}}). \tag{4}$$

2.1 Incremental Value Iteration

Because the state and option spaces may be large, it is important that the sweeps over them can be performed incrementally. For the sweeps over states this is immediate; the states can be updated one-by-one in any order with graceful degradation (and typically improvements if the ordering can be done intelligently). The maximum over actions in (4) can also be done incrementally by keeping track for each state of the option that is currently thought to be best in that state. Let us assume this is done with some sort of function approximator, so that $o(s)$ is an estimate of the best option to take in s. Then, to do an incomplete sequence of planning updates for state s, analogous to the value iteration given above, one does:

Incremental Value Iteration

Input: state s to be updated
Requires: model $\hat{r}, \hat{p}$; value-function approximation $v_{\mathbf{w}}$

$besto \leftarrow o(s)$
$bestv \leftarrow b(s, besto, v_{\mathbf{w}})$
While there is time for some incremental planning, do:
    $o \leftarrow$ some option available in s
    $v \leftarrow b(s, o, v_{\mathbf{w}})$
    If $v > bestv$ then $besto \leftarrow o; bestv \leftarrow v$
end while
$o(s) \leftarrow besto$
$v_{\mathbf{w}}(s) \leftarrow bestv$

where all the arrows indicate assignments, except in the last two steps, where the arrows indicates storage in the function approximators for the approximate best option and approximate value function.

The algorithm above is incremental in the sense that it can consider a limited number of options in a limited amount of time. It is an *anytime* method in the sense that it will give better results the more planning time it is given, but will still do ok if not given much time, even eventually finding the optimum as it is allocated an infinite amount of time over an infinite number of invocations. A remaining problem is that the backed-up value computations (3) still involve a sum over states, and thus naively a loop over all states, which would be impractical. Next we consider way of avoiding this inner loop by taking a closer look at what we might want in the output of the model.

3 The Dynamics Part of a Model

What are the inputs and outputs of a model? The inputs are a state and an option. The outputs are the expected reward while the option executes and something about the state that results when the option stops. That something is a key choice point in the design of a planning agent. Recall that we are assuming that states are represented by numerical feature vectors. Should the model produce the distribution of those result-state feature vectors, a sample feature vector, or the expected value of the feature vector? Below we discuss each of these three possibilities in turn.

3.1 Distribution and Sample Models

So far we have assumed that the model returns, in the function p, the probability of each possible resultant state at termination of the option. Such a model must be able to produce the entire distribution of resultant states, and accordingly we call it a *distribution model*. Distribution models are important theoretically and have been used effectively when the possible distributions can be assumed to be of a special form (e.g., Gaussian), as in the PILCO method (Deisenroth and Rasmussen, 2011).

However, for general distributions we regard this strategy as impractical because distributions are such potentially large objects. If the state is represented by a feature vector of dimension d, then the first moment of its distribution is a d-vector, but the second moment is a $d \times d$ matrix, and the third moment is $d \times d \times d$, and so on. Unless the distribution can be assumed to be of a known special form, these are very large objects. In general, all the moments are required to characterize the distribution, but that would be hopelessly unwieldy for large d. Even if we had the full distribution, we would still have the problem of computing the summation over possible result states in (3). This would seem impossible in general; only if the distribution has a very special form can it be done computationally efficiently

We consider it vitally important not to adopt an approach that is limited in its generality. To do otherwise is not to heed the "bitter lesson" from the history of AI (outlined in Section 1 of Sutton et al. 2018) and thus to risk irrelevance as computational power scales exponentially. If a distribution model is to be used, one would need an efficient way to store it and compute with it. We do not at present see how this could be done while retaining applicability and scalability for arbitrary dynamical environments.

A related idea that is more practical in some ways is for the model to return not the full distribution, but a sample from it. Then a sense of the full distribution can be obtained incrementally by repeated sampling. Such a *sample model* is particularly well suited for sample-based planning methods such as Monte Carlo Tree Search. The first advantage of sampling models is that they can use arbitrary distributions to generate the samples, yet still produce objects of a standard, limited size (e.g., feature vectors of dimension d). A second advantage is that it is easier to iterate them, that is, to take the state output of the model's projection and feed it in again as the state input, projecting to a sample state two iterations later, and so on. The first disadvantage of sampling models is that they are stochastic, unlike distribution models, which are deterministic. To use a sampling model creates an additional branching factor in planning, as multiple samples need to be drawn to gain a representative sense of what might happen.

A second disadvantage of sample models is that, although we don't need to produce the whole distribution as output, we may have to represent its full complexity in order to sample from it. This would seem to make sample models suffer from the same problem as distribution models: an unacceptable loss of generality and scalability with computation. Perhaps there is some approximation strategy by which one could obtain good approximate results limited only by computational resources. We have not yet found such a strategy for distribution or sample models. We turn now to a third approach for which such a strategy clearly exists.

3.2 Expectation Models

The third approach is for the model to produce the *expectation* of the result-state feature vector. This kind of model we call an *expectation model*. The advantages of expectation models are that the state output is compact (like a sample model) and deterministic (like a distribution model).

The apparent disadvantage of an expectation model is that it is only a partial

characterization of the distribution. For example, if the result of an action (or option) is that two binary state features both occur with probability 0.5, but are never present (=1) together, then an expectation model can capture the first part (each present with probability 0.5), but not the second (never both present). This may not be a substantive limitation, however, as we can always add a third binary state feature, for example, for the AND of the original two features, and then capture the full distribution with the expectation of all three state features.

It may seem that the expectation model puts more pressure on the state representation, and this is true, but it is also a substantial relaxation of what is required from a distribution and a sample model. Both of these other models have to find and represent any correlations or anti-correlations between the state features, much as an expectation model would have to. Overall it is not clear that the amount of work required changes when moving from a distribution or sampling model to an expectation model; more likely the burden is just shifted from constructing the sample/distribution model to constructing the state representation. In general, the state representation (feature vector) can be expected to be larger (to have more components) with an expectation model.

3.3 Expectation Models and Linearity

An important observation is that if the value function is linear in the state features, then there is no loss of generality when using an expectation model in planning. It is easy to show this formally, starting from planning by value iteration as given earlier by (2). For an approximate value function to be linear in the state features means that it is of the form $v_{\mathbf{w}}(s) \doteq \mathbf{w}^\top \mathbf{x}(s)$ where $\mathbf{x}(s)$ is the feature vector corresponding to state s. Plugging this into the general distribution-model form of value iteration (1) yields

$$
\begin{aligned}
v_{\mathbf{w}}(s) \doteq \mathbf{w}^\top \mathbf{x}(s) &\leftarrow \max_o \left[\hat{r}(s,o) + \sum_{s'} \hat{p}(s'|s,o) v_{\mathbf{w}}(s') \right] && \text{(from (2))} \\
&= \max_o \left[\hat{r}(s,o) + \sum_{s'} \hat{p}(s'|s,o) \mathbf{w}^\top \mathbf{x}(s') \right] \\
&= \max_o \left[\hat{r}(s,o) + \left(\sum_{s'} \hat{p}(s'|s,o) \mathbf{x}(s') \right)^\top \mathbf{w} \right] \\
&= \max_o \left[\hat{r}(s,o) + \mathbf{w}^\top \hat{\mathbf{x}}(s,o) \right] && (5)
\end{aligned}
$$

where $\hat{\mathbf{x}}(s,o)$ is the expected feature vector if option o is initiated in state s. The pair $(\hat{r}, \hat{\mathbf{x}})$ is an expectation model. That the last line is equal to the first proves that no generality over a distribution model has been lost by the use of the expectation model, $(\hat{r}, \hat{\mathbf{x}})$.

The same equations also shows that, if we are using an expectation model, then in general the approximate value function must be linear, as (5) can only equal (from (2)) for general p if $v_{\mathbf{w}} : \mathbb{R}^d \to \mathbb{R}$ is linear.

Finally, if the approximate value function is linear in the state features, then planning with a nonlinear model offers no improvement over a linear expectation model (, Li, Taylor, Painter-Wakefield, & Littman 2008). This follows because we can compute analytically the linear value function that we get by linear TD(0) with the real system, and we can compute analytically the linear value function we get by the least-squares linear expectation model (with the same features), and these two are the same (analytically). Expectation models seem to imply that not only that the value function is linear, but that the model itself can and therefore ought to be linear. The linear-model case has many advantages, including convergence of planning to the same solution independent of the search-control distribution and the option of using a variation of prioritized sweeping (Sutton, Szepesvari, Geramifard, & Bowling, 2008).

In these senses, the choices of an expectation model, a linear approximate value function, and a linear model are tightly linked. It is not a rigorous argument that one must make any one of these choices on its own, but once some have been made, there are strong reasons to adopt all of them. Let us call planning based on all three choices expectation-based planning.

4 GVF Learning Algorithms

In the rest of this document we consider several special cases of general value functions (GVFs). Rather than giving different learning algorithms for each, we present here a general learning algorithm that will apply to all of them. The standard GVF for an arbitrary target policy $\pi : \mathcal{A} \times \mathcal{S} \to [0,1]$, target stopping condition $\beta : \mathcal{S} \to [0,1]$, and cumulant $C_t \in \mathbb{R}$ is

$$v_{\pi,\beta,C}(s) \doteq \sum_{k=t}^{\infty} \mathbb{E}\left[C_{k+1} \prod_{j=t+1}^{k} \big(1-\beta(S_j)\big) \;\middle|\; S_t{=}s, A_{t:k} \sim \pi \right] \tag{6}$$

$$= \mathbb{E}\big[C_{t+1} + \big(1-\beta(S_{t+1})\big) v_{\pi,\beta,C}(S_{t+1}) \;\big|\; S_t{=}s, A_t \sim \pi \big] \tag{7}$$

$$= \mathbb{E}\Big[\rho_t \Big(C_{t+1} + \big(1-\beta(S_{t+1})\big) v_{\pi,\beta,C}(S_{t+1}) \Big) \;\Big|\; S_t{=}s, A_t \sim b \Big], \tag{8}$$

where $\rho_t = \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$ is an importance sampling ratio, with behavior policy $b : \mathcal{A} \times \mathcal{S} \to [0,1]$, to accomodate the possibility of off-policy training. The general online algorithm for learning the weight vector $\mathbf{w} \in \mathbb{R}^d$ in a linear approximation $\mathbf{w}^\top \mathbf{x}(s) \approx v_{\pi,\beta,C}(s)$ is

$$\begin{aligned} \mathbf{w}_{t+1} &\doteq \mathbf{w}_t + \alpha \delta_t \mathbf{z}_t \\ \delta_t &\doteq C_{t+1} + (1-\beta_{t+1})\, \mathbf{x}_{t+1}^\top \mathbf{w}_t - \mathbf{x}_t^\top \mathbf{w}_t \\ \mathbf{z}_t &\doteq \rho_t \left((1-\beta_t)\, \lambda_t \mathbf{z}_{t-1} + \mathbf{x}_t \right) \end{aligned} \tag{9}$$

where $\alpha < 0$ is a step-size parameter, β_t is shorthand for $\beta(S_t)$, $\mathbf{z}_t \in \mathbb{R}^d$ is an eligibility-trace vector, and $\lambda_t \in [0,1]$ is some degree of bootstrapping at step t. This is a naive (semi-gradient) eligibility-trace update algorithm; we could replace it with any of the more sophisticated off-policy methods such as V-trace or Emphatic TD(λ) to obtain a more robust (albeit slightly more complicated) algorithm.

As an example, let us apply this general algorithm to learning the value function for our discovery agent. The agent operates in the average-reward setting, as is appropriate in the control case with function approximation. We use the *undiscounted* return,

$$G_t \doteq \sum_{k=1}^{\infty} \big(R_{t+k} - r(b)\big), \tag{10}$$

which is defined with respect to the long-term average reward of the agent's behavior policy b,

$$r(b) \doteq \lim_{k\to\infty} \frac{1}{k} \sum_{t=1}^{k} \mathbb{E}[R_t \mid A_{0:t-1} \sim b]\,. \tag{11}$$

The differential state-value function is then defined in the usual way with respect to the return, and approximated as a linear function of the features:

$$v_b(s) \;\doteq\; \mathbb{E}[G_t \mid S_t{=}s, A_{t:\infty} \sim b] \;\;\approx\;\; \mathbf{v}^\top \mathbf{x}(s), \tag{12}$$

where $\mathbf{v} \in \mathbb{R}^d$ is a weight vector. To learn it on-policy using the general algorithm (9) we simply substitute $\mathbf{v}$ for $\mathbf{w}$ and choose $\beta_t \doteq 0$, $\rho_t = 1$, and $C_t \doteq R_t - \bar{R}_{t-1}$, where $\bar{R}_t$ is an approximation to $r(b)$ that is separately learned by the rule:

$$\bar{R}_{t+1} \doteq \bar{R}_t + \bar{\alpha}\left(R_{t+1} - \bar{R}_t + \mathbf{v}_t^\top \mathbf{x}_{t+1} - \mathbf{v}_t^\top \mathbf{x}_t\right). \tag{13}$$

5 The Cycle of Discovery

A key remaining question is how to come up with options whose models are useful in planning. An appealing strategy is to consider options that solve subproblems or auxiliary tasks of some sort. For example, subproblems have been proposed to control the individual pixels of an image (Jaderberg et al., 2016), to reach certain states (Mirowski et al., 2016), and to learn aspects of the distribution of reward (Bellemare, Dabney, and Munos, 2017). A weakness of such these proposed subproblems is that they ignore the reward (or, equivalently, optimize a reward or cumulant different from the main task's reward).

The solutions to these previously considered subproblems are valid options, but not options designed to obtain high main-task reward, and thus they are unlikely to be useful in planning. To see this more explicitly, recall that planning is by iteratively applying a value iteration process (4), at various states s, to improve an approximate value function $v_{\mathbf{w}}$:

$$v_{\mathbf{w}}(s) \leftarrow \max_o b(s, o, v_{\mathbf{w}}),$$

where $b(s, o, v_{\mathbf{w}})$ is a backed-up value, defined in the case of expectation models as

$$b(s, o, v_{\mathbf{w}}) = \hat{r}(s, o) + \mathbf{w}^\top \hat{\mathbf{x}}(s, o),$$

where $\hat{r}(s,o)$ is the expected cumulative reward and $\hat{\mathbf{x}}(s,o)$ is the expected next state (all for the main task; these are unaffected by the subproblem) as defined earlier. A prospective option o can only be useful in state s if its backed-up value $b(s,o,v_{\mathbf{w}})$ is large (so that it takes the maximum in planning in the equation two above, or improves in the incremental planning strategy). This is unlikely to happen for a random option, or for an option solving a subproblem with an arbitrary reward, or for an option that highly values stopping in states in which the real value is low. Instead, subproblems should include the main reward signal and the main value function in their definitions.

In Section 5.2 we propose subproblems based on achieving of each of the state features with high reward. In particular, for state feature i, we propose the subproblem whose solution is an option (π^i, β^i) that maximizes for each state s the expected value of the cumulative reward from s (following π^i) until stopping in s' (according to β^i), plus $v_{\mathbf{w}}(s') + \kappa^i x^i(s')$, where κ^i is an appropriately chosen bonus for reaching a state in which feature i is high (assuming $\kappa^i > 0$). If the bonus is 0, then the subproblem would be identical to the main task and would not be expected to produce a useful option (because the option learned would be the same as the base policy).

5.1 Feature-based Subproblems

We propose subproblems based on achieving of each of the state features with high reward. In particular, for state feature i, we propose the subproblem whose solution is an option (π^i, β^i) that maximizes for each state s the expected value of the cumulative reward from s (following π^i) until stopping in s' (according to β^i), plus $v_{\mathbf{w}}(s') + \kappa^i x^i(s')$, where κ^i is an appropriately chosen bonus for reaching a state in which feature i is high (assuming $\kappa^i > 0$). If the bonus is 0, then the subproblem would be identical to the main task and would not be expected to produce a useful option (because the option learned would be the same as the base policy). Solving such subproblems yields options that might result in improvements during planning.

We discuss the feature-based subproblems more thoroughly in Section 3.4 below.

5.2 Feature-based Subproblems: From Features to Options

For each feature i we define a subproblem of finding an option (π^i, β^i) that maximizes the return plus a bonus proportional to the value of the feature at the stopping state. The prediction part of this subproblem is to approximate the value function

$$v^i(s) \doteq \mathbb{E}_{\pi^i}\big[G_t + \kappa^i x^i_K \;\big|\; S_t = s, K \sim \beta^i\big], \tag{14}$$

where K is the time of stopping (a random variable), and κ^i is a scalar factor on a bonus for stopping with feature i present. The control part of this subproblem is to find the option (π^i, β^i) that maximizes this value.

For tenured features we allocate resources to solving their subproblems. We take an actor-critic approach with learned weights for a linear approximate value function as well as weights for an option's policy π^i (from which will be derived its stopping condition β^i). The goal for the value weights $\mathbf{v}^i \in \mathbb{R}^d$ is that the linearly approximated

value of each state s approximates

$$\begin{aligned}\mathbf{x}(s)^\top \mathbf{v}^i &\approx v^i(s) \\ &\doteq \mathbb{E}\big[R_1 - r(b) + \cdots + R_K - r(b) + v_\pi(S_K) + \kappa^i x^i_K \;\big|\; S_0 = s, A_{0:K-1} \sim \pi^i, K \sim \beta^i\big] \\ &\approx \mathbb{E}\big[R_1 - \bar{R} + \cdots + R_K - \bar{R} + \mathbf{x}_K^\top \mathbf{v} + \kappa^i x^i_K \;\big|\; S_0 = s, A_{0:K-1} \sim \pi^i, K \sim \beta^i\big]\,,\end{aligned} \tag{15}$$

where $\bar{R}$ is a learned approximation to $r(b)$. This goal is in the standard GVF form with policy and horizon given by π^i and β^i, and with cumulant:

$$C^i_{t+1} \doteq R_{t+1} - \bar{R}_t + \beta^i_{t+1}\Big(\mathbf{x}^\top_{t+1}\mathbf{v}_t + \kappa^i x^i_{t+1}\Big). \tag{16}$$

Thus we can apply the general rule (9) with $\mathbf{v}^i$ substituting for $\mathbf{w}$, π^i for π, β^i for β, and C^i for C. The result is a not only a sequence of weight vectors $\{\mathbf{v}^i_t\}$, for each feature i, but also a sequence of TD errors, which we denote as $\{\delta^i_t\}$, and a sequence of eligibility-trace vectors, which we denote by $\{\mathbf{z}^i_t\}$.

It remains to specify how feature i's option's policy and stopping condition are learned. These need not be linear in the state features. Let the weight vector of the policy be denoted by $\boldsymbol{\theta}^i$ and the policy by $\pi(a|s, \boldsymbol{\theta}^i)$. The policy weight vector is updated by the actor–critic policy-gradient rule:

$$\boldsymbol{\theta}^i_{t+1} \doteq \boldsymbol{\theta}^i_t + \alpha\delta^i_t \mathbf{z}^i_t \tag{17}$$

$$\mathbf{z}^i_t \doteq (1 - \beta^i_t)\lambda^i_t \rho^i_t \mathbf{z}^i_{t-1} + \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta}^i), \tag{18}$$

where δ^i_t comes from the learning of $\mathbf{v}_i$, and α, $\mathbf{z}^i_t$, all nd λ^i_t play similar roles as the similarly named quantities there but are distinct and may take on different values.

The stopping condition of feature i's option could also be learned, with its own weight vector, but it may be possible to just determine stopping in the moment. There would seem to be a clear ideal condition for stopping:

$$\beta^i_{t+1} \doteq \begin{cases} 1 & \text{if } v_b(S_{t+1}) + \kappa^i x^i_{t+1} \geq v^i(S_{t+1}); \\ 0 & \text{otherwise.} \end{cases} \tag{19}$$

which is naturally approximated as

$$\beta^i_{t+1} \doteq \begin{cases} 1 & \text{if } \mathbf{x}^\top_{t+1}\mathbf{v}_t + \kappa^i x^i_{t+1} \geq \mathbf{x}^\top_{t+1}\mathbf{v}^i_t; \\ 0 & \text{otherwise.} \end{cases} \tag{20}$$

This is an elegant way of determining termination, but it is just one possibility. Alternatively, a separate set of weights could be used and tuned towards this sort of objective. There are also intermediate possibilities, such as favoring the above condition, but modulating it with some learned weights. Experimentation is needed to determine the best choice.

5.3 From Options to Environmental Models

When a state feature is tenured, we not only learn an option for achieving it, but also an approximate model of that option. This model takes in a starting state-feature

vector and produces an estimate of the expectation of the state-feature vector when the option stops (and an estimate of the expected cumulative reward along the way). This is known as an *expectation* model, to distinguish it from a full *distribution* model, which produces the probability of each resulting feature vector in some sense, and from a *sample* model, which produces a sample feature vector from that distribution. The process of going from input feature vector to output expected feature vector via the model is termed *projection*.

On the output side of projection, we think of each predicted (expected) tenured feature as being produced individually. The model of tenured feature i's option, (π^i, β^i), for each output feature j, is an estimate of the expected value:

$$\mathbb{E}\big[x^j_K \;\big|\; S_t{=}s, A_{t:K-1} \sim \pi^i, K \sim \beta^i\big] \approx \mathbf{x}(s)^\top \mathbf{m}^{ij}_t, \tag{21}$$

where $\mathbf{m}^{ij}_t$ is a vector of learned weights. It may not be obvious, but this form is also a special case of a GVF, and the weight vector can be learned by our standard algorithm (9) with $\pi \doteq \pi^i$, $\beta \doteq \beta^i$, and $C_t \doteq (1-\beta^i_t)x^j_t$. Similarly, the reward part of the option model, the expected value of the cumulative differential reward during the option's execution,

$$\begin{aligned}
&\mathbb{E}\big[R_1{-}r(b) + \cdots + R_K{-}r(b) \;\big|\; S_0{=}s, A_{0:K-1} \sim \pi^i, K \sim \beta^i\big] && (22)\\
&\quad\approx \mathbb{E}\big[R_1{-}\bar{R} + \cdots + R_K{-}\bar{R} \;\big|\; S_0{=}s, A_{0:K-1} \sim \pi^i, K \sim \beta^i\big] && (23)\\
&\quad\approx \mathbf{x}(s)^\top \mathbf{r}^i_t, && (24)
\end{aligned}$$

where $\mathbf{r}^i_t$ is a weight vector, can be learned as a GVF with the standard algorithm (9) with $\pi \doteq \pi^i$, $\beta \doteq \beta^i$, and $C_t \doteq R_t - \bar{R}_{t-1}$ where $\bar{R}$ is learned as before with (13).

Projection generally does not operate with whole feature vectors, but only with their tenured features. The missing untenured features are effectively treated as zeros, which is reasonable because our models are linear in the input features, and tenured features are selected based on the extent to which they to contribute to successful predictions. If all of a feature's outgoing weights are zero, then the feature will be untenured *and* can safely be ignored without affecting any predictions. Similarly on the output side, there is no need to produce an expected feature value for untenured features that have no subsequent consequences.[1] This is an important economy. Let $m \ll n$ be the numbered of tenured features. Then instead of n^3 weights, only $m^2 n$ need be learned, and only $m(m+1)$ are involved in a single projection step.

5.4 Curation

In this section we have presented just the outlines of an approach to discovery. One aspect that should be considered incomplete is the specification of how the various pieces, such as features, options, and models, are monitored, assessed, and organized to make the overall system work as well as possible. This activity in general is referred

[1]Here we are referring to projections—to the use of the model rather than to its learning. When learning the model we must consider all the features in case they are important but that has not yet been realized.

to as *curation*.[2] As just one example, how do we decide which option's models to consider in planning? This choice is sometimes referred to as *search control*. If search control is done well, then planning may contribute greatly to computing the main value function and a good policy in a timely manner. However, if search control is done poorly, then all the planning could be a waste of time, or worse. Statistics of various sorts are presumably computed and maintained on each option and situation in which its model might be used in planning, and then these statistics are used to inform search control.

6 Generalizing Planning to Option Values and Option Models

So far we have described planning as the use of option models to compute better approximations, $v_{\mathbf{w}}$, of the main value function, v_π. Later, as part of the discovery process, we introduced additional value functions v^i, one for each feature/option (14) (each approximated with a feature vector $\mathbf{v}^i$). It is natural then to extend the idea of planning to better approximate these value functions as well. In fact, all the equations as we used in the original planning section (Section 2) apply to this case as well if the parameter vector $\mathbf{w}$ is reinterpreted as possibly being the per-feature parameter vector $\mathbf{v}^i$. Thus, if we have the right models, we may be able to plan option value functions as well as learn them.

Moreover, this idea can be carried one big step further. We mentioned at the beginning of Section 2 and detailed in Subsection 5.3 how the estimation of each option's model can be interpreted as learning a set of $n+1$ GVFs. Each of these GVFs can not only be learned with the generic value function algorithm given in Subsection 4, but they can also be planned, again using the planning equations in Section 2 reinterpreted generically. In this way we can not only learn the models, but also plan to them.

Mindblowing.

All this planning towards lower level goals is only possible if we have the appropriate models. Low-level models, even one-step models, are particularly useful here. One would like to proceed with whatever models one has, to whatever extent they are trustworthy. Curation would assess trustworthiness; curation would be crucial in directing the planning process. If it was done right, one imagines the process would work naturally upwards from whatever level could first be learned and trusted.

7 Logistic Planning

The base planning system suggested in Section 2 is one based on value functions and expectation models that are both linear in the state-feature representation. An argument is made that once one has committed to an expectation model, then only a linear value function is guaranteed not to lose generality. Then, an argument is

[2]The dictionary on my computer defines "to curate" as "to select, organize, and look after the items in (a collection or exhibit)".

made that once one is committed to a linear value function, there is no further loss by commiting to linear models. The two linearities are strongly interlinked and reinforcing.

But is there no other choice? Sometimes the linearity seem over-constraining. Suppose we have binary features. Then the transition part of the model should always produce values between zero and one, and it would be natural to force this by applying a squashing non-linearity. In such a case, perhaps we don't want to produce the expected next feature vector, as that would suggest a squared-error criterion. Maybe we want a cross-entropy criterion? We would like to extend the result in Section 3.3—that planning with an expectation model loses nothing if the value function is linear—to other settings, such as with squashed nonlinearities, or at least with the model itself involving squashed nonlinearities. We are still searching for this but have not found it yet.

In the meantime, we are going ahead with theorizing about feature-based models with binary features and approximate transition dynamics with logistic nonlinearities. We have a theorem wish seems to argue mildly against it. The theorem says that if the value function is linear, then the model might as well be linear, because infinite planning the best linear model (fit to a batch of data) returns the value function as experiencing that data repeatedly and learning in a model-free manner from it. This suggests the making both value functions and models linear, but it does not really require it. Which is good because this case seems potentially prone to extrapolation and explosion.

So maybe we just need to decline going the linear-linear way. The values can be capped, at least at the lower end. The models can be logistic, capturing the fact that actual state features may be restricted to zero and one. If they are, then I believe there is a sense in which the result in Section 3.3 still holds.

References

Bellemare, M. G., Dabney, W., Munos, R. (2017). A distributional perspective on reinforcement learning. arXiv preprint arXiv:1707.06887.

Deisenroth, M., Rasmussen, C. E. (2011). PILCO: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on machine learning* (ICML-11) (pp. 465–472).

Jaderberg, M., Mnih, V., Czarnecki, W. M., Schaul, T., Leibo, J. Z., Silver, D., Kavukcuoglu, K. (2016). Reinforcement learning with unsupervised auxiliary tasks. ArXiv:1611.05397.

Mahmood, A. R. (2017). *Incremental Off-Policy Reinforcement Learning Algorithms.* Ph.D. thesis, University of Alberta, Edmonton.

Mirowski, P., Pascanu, R., Viola, F., Soyer, H., Ballard, A. J., Banino, A., ..., Kumaran, D. (2016). Learning to navigate in complex environments. arXiv preprint arXiv:1611.03673.

Modayil, J., Sutton, R. S. (2014). Prediction driven behavior: Learning predictions

that drive fixed responses. In *AAAI-14 Workshop on Artificial Intelligence and Robotics*, Quebec City, Quebec, Canada.

Parr, R., Li, L., Taylor, G., Painter-Wakefield, C., Littman, M. L. (2008). An analysis of linear models, linear value-function approximation, and feature selection for reinforcement learning. In *Proceedings of the 25th international conference on Machine learning* (pp. 752–759). ACM.

Sutton, R. S., Szepesvári, Cs., Geramifard, A., Bowling, M., (2008). Dyna-style planning with linear function approximation and prioritized sweeping. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence*, pp. 528–536.

Sutton, R. S., and the Core RL working group. (2018). Initial Report of the Working Group on Core Reinforcement Learning. DeepMind Report.

Veeriah, V., Zhang, S., Sutton, R. S. (2017). Crossprop: Learning representations by stochastic meta-gradient descent in neural networks. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases* (pp. 445–459). Springer.