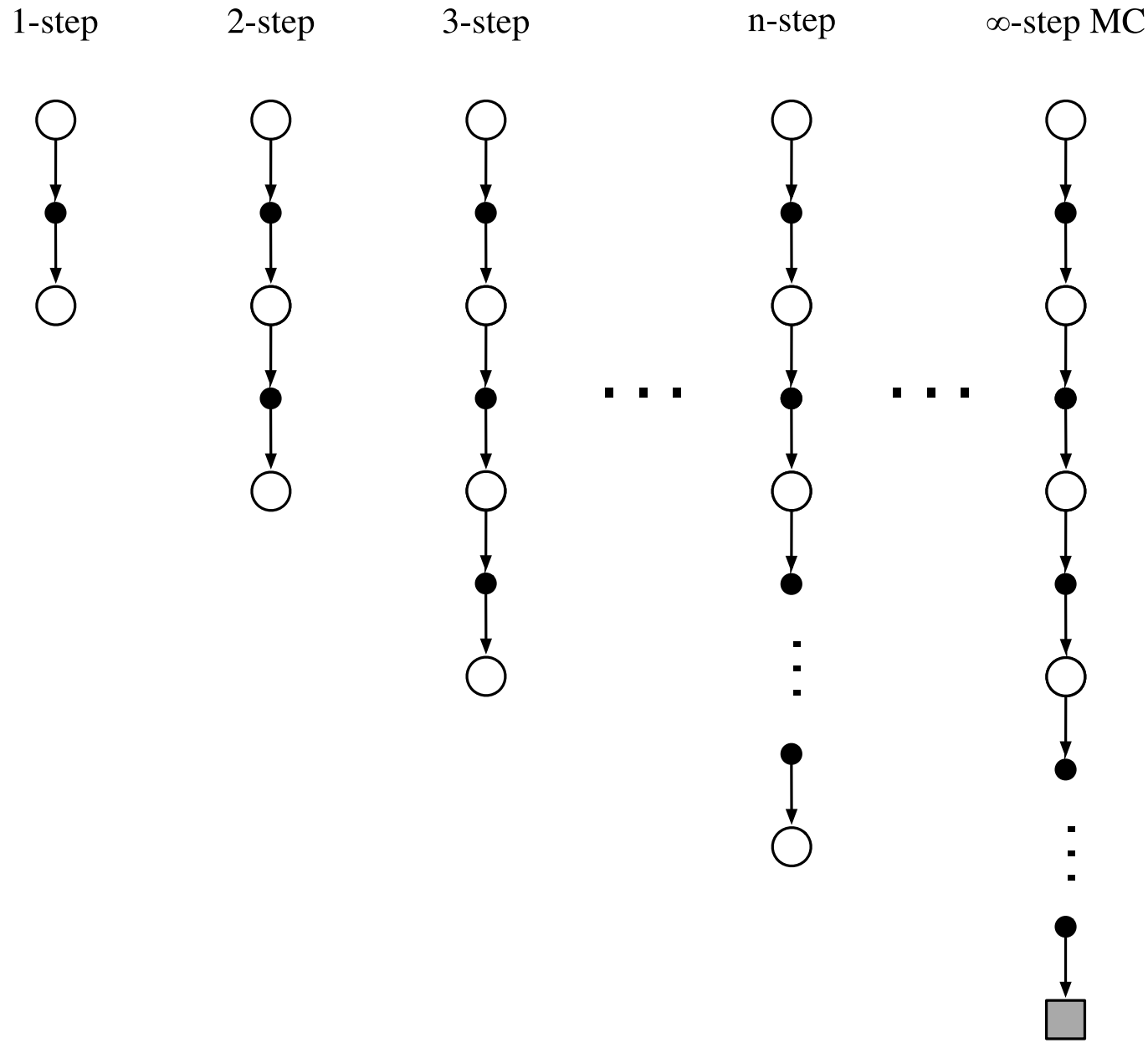
Temporal Abstraction in Reinforcement Learning

Pierre-Luc Bacon and Doina Precup

February 17, 2017

Multi-steps Boostrapping



Multi-steps Boostrapping

" [...] they free you from the tyranny of the time step." —

In S&B 2017, chapter 7

Multi-steps Boostrapping: key idea

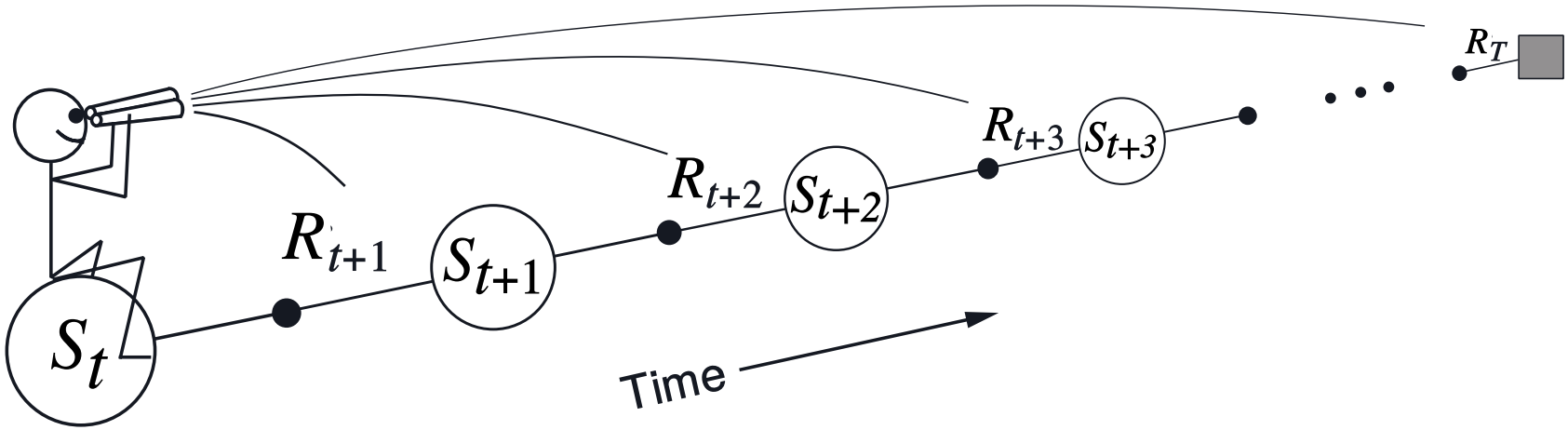
The n-step return estimator is defined as:

$$G_t^{(n)} = R_t + \gamma R_{t+1} + \ldots + \gamma^{n-1} R_{t+n-1} + \gamma^n V_{t+n-1}(S_{t+n})$$

we then use it as an **update target** for TD:

$$V_{t+n}(S_t) = V_{t+n-1} + \alpha \left(G_t^{(n)} - V_{t+n-1}(S_t) \right)$$

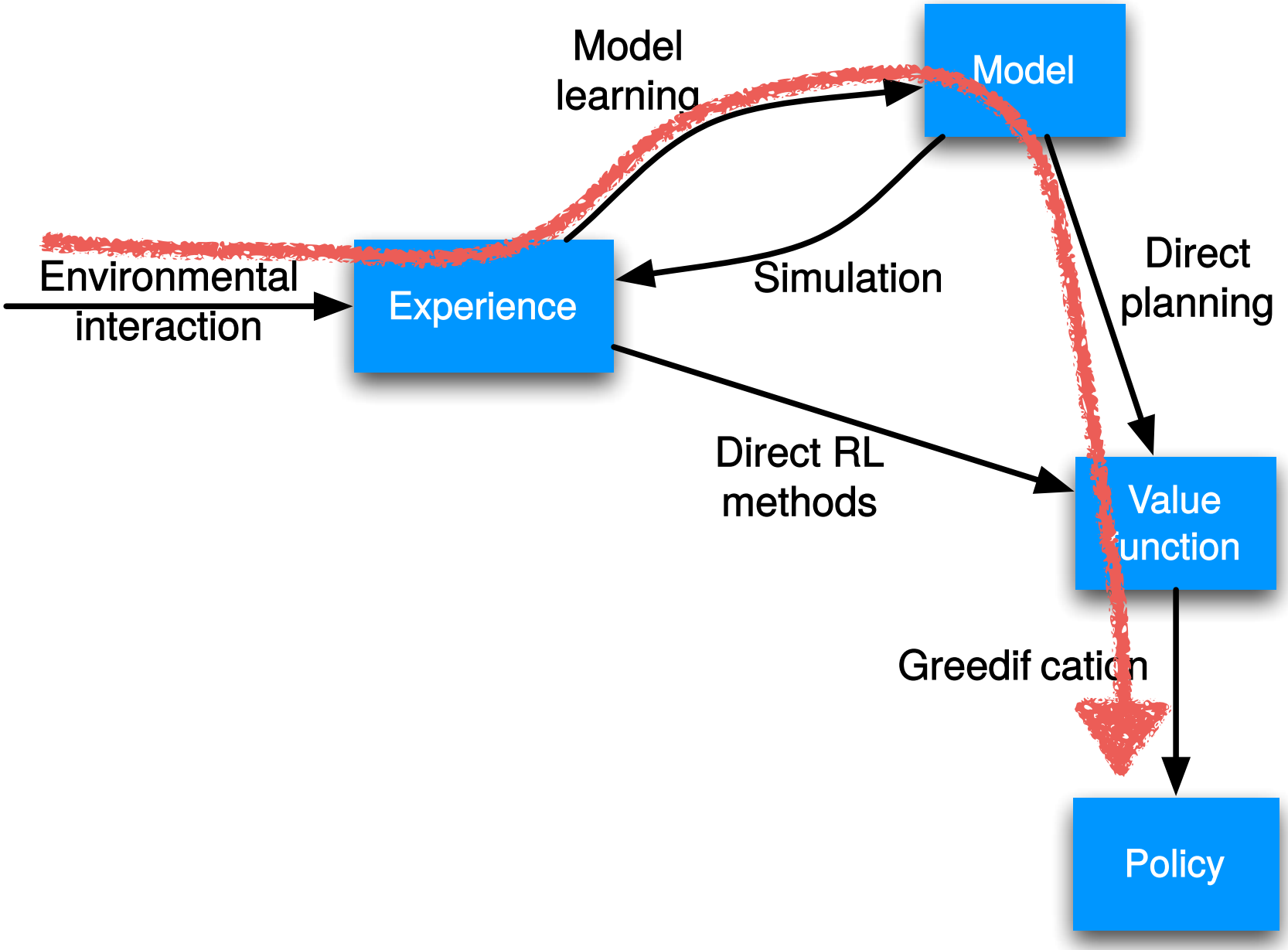
Forward view



The n-step return estimator depends on future rewards, despite being defined at time t. How do we deal with that ?

1. Wait (batch updating)
2. Use eligibility traces (we'll see this in a few weeks)
3. Use n-steps models

Model-based RL: recap



Planning: indirect RL

Key idea

We learn models from data, and plug them back into our Bellman equations.

Remember: with the true r_π and P_π, we have:

$$v = r_\pi + \gamma P_\pi v$$

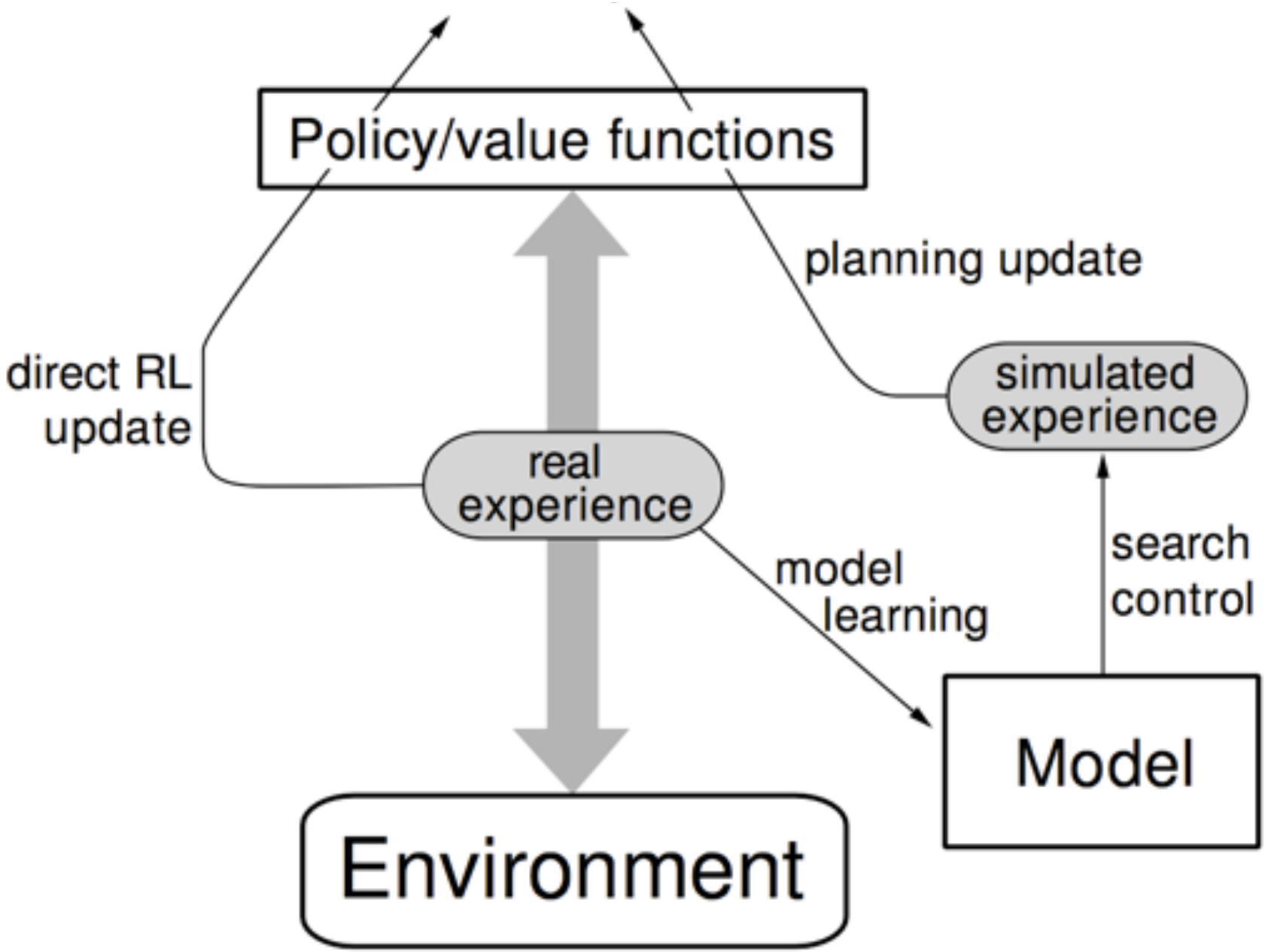
Let b be an approximation of r_π and F the model for P_π:

$$v = b + \gamma F v$$

We can then solve for v with :

1. DP methods
2. Sample-based TD methods (Dyna belongs here)

Dyna


Policy/value functions
planning update
direct RL
update
simulated
experience
real
experience
search
control
model
learning
Model
Environment

Sutton 1995: n-steps models

As appeared in: *Proceedings of the 12th Int. Conf. on Machine Learning*, 531–539, Morgan Kaufmann, 1995.

TD Models: Modeling the World at a Mixture of Time Scales

Richard S. Sutton
Stow Research
sutton@gte.com

Abstract

Temporal-difference (TD) learning can be used not just to predict *rewards*, as is commonly done in reinforcement learning, but also to predict *states*, i.e., to learn a model of the world's dynamics. We present theory and algorithms for intermixing TD models of the world at different levels of temporal abstraction within a single structure. Such multi-scale TD models can be used in model-based reinforcement-learning architectures and dynamic programming methods in place of conventional Markov models. This enables planning at higher and varied levels of abstraction, and, as such, may prove useful in formulating methods for hierarchical or multi-level planning and reinforcement learning. In this paper we treat only the *prediction* problem—that of learning a model and value function for the case of fixed agent behavior. Within this context, we establish the theoretical foundations of multi-scale models and derive TD algorithms for learning them. Two small computational experiments are presented to test and illustrate the theory. This work is an extension and generalization of the work of Singh (1992), Dayan (1993), and Sutton & Pinette (1985).

1 Multi-Scale Planning and Modeling

Model-based reinforcement learning offers a potentially elegant solution to the problem of integrating planning into a real-time learning and decision-making agent (Sutton, 1990; Barto et al., 1995; Peng & Williams, 1993, Moore & Atkeson, 1994; Dean et al., in prep). However, most current reinforcement-learning systems assume a single, fixed time step: actions take one step to complete, and their immediate consequences become available after one step. This makes it difficult to learn and plan at different time scales. For example, commuting to work involves planning at a high level about which route to drive (or whether to take the train) and at a low level about how to steer, when to brake, etc. Planning is necessary at both levels in order to optimize precise low-level movements without becoming lost in a sea of detail when making decisions at a high level. Moreover, these levels cannot be kept totally distinct and separate. They must interrelate at least in the sense that the actions and plans at a high levels must be turned into actual, moment-by-moment decisions at the lowest level.

The need for hierarchical and abstract planning is a fundamental problem in AI whether or not one uses the reinforcement-learning framework (e.g., Fikes et al., 1972; Sacerdoti, 1977; Kuipers, 1979; Laird et al., 1986; Korf, 1985; Minton, 1988; Watkins, 1989; Drescher, 1991; Ring, 1991; Wixson, 1991; Schmidhuber, 1991; Tenenberg et al., 1992; Kaelbling, 1993; Lin, 1993; Dayan & Hinton, 1993; Dejong, 1994; Chrisman, 1994; Hansen, 1994; Dean & Lin, in prep). We do not propose to fully solve it in this paper. Rather, we develop an approach to multiple-time-scale modeling of the world that may eventually be useful in such a solution. Our approach is to extend temporal-difference (TD) methods, which are commonly used in reinforcement learning systems to learn value functions, such that they can be used to learn world models. When TD methods are used, the predictions of the models can naturally extend beyond a single time step. As we will show, they can even make predictions that are not specific to a single time scale, but intermix many such scales, with no loss of performance when the models are used. This approach is an extension of the ideas of Singh (1992), Dayan (1993), and Sutton & Pinette (1985).

Most prior work on multi-scale modeling has focused on state abstraction: Which sets of states can be treated as a group? What variables can be ignored? What is a good form of generalization between states? In this paper we instead focus exclusively on the relatively ignored *temporal* aspects of abstraction. In fact, we will assume each state is recognized and treated as

Bellman equations for 2-steps models

Let's expand v once:

$$\begin{aligned} v &= r_\pi + \gamma P_\pi v \\ &= r_\pi + \gamma P_\pi (r_\pi + \gamma P_\pi v) \\ &= r_\pi + \gamma P_\pi r_\pi + \gamma^2 P_\pi^2 v \end{aligned}$$

Define the **2-step reward model** as:

$$b^{(2)} \doteq r_\pi + \gamma P_\pi r_\pi$$

and the **2-step transition model**:

$$F^{(2)} \doteq \gamma^2 P_\pi^2$$

The Bellman equation can then also be written as:

$$v = b^{(2)} + F^{(2)} v$$

(The gamma term is folded in F. A matter of taste...)

n-steps models

$$b^{(n)} \doteq \sum_{t=0}^{n-1} (\gamma P_\pi)^t r_\pi \qquad\qquad F^{(n)} \doteq (\gamma P_\pi)^n$$

And once again, the Bellman equations still hold:

$$v = b^{(n)} + F^{(n)} v$$

Question : what happens as $n \to \infty$?

Fun fact: model composition

Homogeneous coordinates

Computer graphics, vision, animation people: this is a familiar idea.

$$\begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} u_{11} & u_{12} & u_{13} & t_x \\ u_{21} & u_2 & u_{23} & t_y \\ u_{31} & u_{32} & u_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

where the u_{ij} are elements of a rotation matrix and $[t_x, t_y, t_z]$ is a translation vector.

Bellman equations in homogeneous form

$$\begin{bmatrix} v \\ 1 \end{bmatrix} = \begin{bmatrix} P_\pi & r_\pi \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ 1 \end{bmatrix} = \begin{bmatrix} P_\pi v + r_\pi \\ 1 \end{bmatrix}$$

where P_π is a block matrix of size $n \times n$ (n being the number of states) and r_π is a column vector of size n.

Composing models

$$M \doteq \begin{bmatrix} \gamma P_\pi & r_\pi \\ 0 & 1 \end{bmatrix}$$

so that:

$$v = Mv$$

A 2-steps models is then:

$$M^{(2)} \doteq M^2 = \begin{bmatrix} \gamma^2 P_\pi^2 & \gamma P_\pi r_\pi + r_\pi \\ 0 & 1 \end{bmatrix}$$

and generally:

$$M^{(n)} \doteq M^n$$

and :

$$v = M^{(n)} v$$

Digression: RNN

With M defined as usual:

$$f(v_k; (r_\pi, \gamma P_\pi)) \doteq Mv$$

You can now think of the iterates of value iteration as the linear dynamical system:

$$v_{k+1} = f(v_k; (r_\pi, P_\pi))$$

f is a recurrent neural net ! Your "hidden state" is v_k. The fixed-point is computed by an "infinitely deep" neural net.

$$v_\pi = f\Big(f\Big(\dots f\big(v_0; (r_\pi, P_\pi)\big); (r_\pi, P_\pi)\Big); (r_\pi, P_\pi)\Big)$$

Let's `T.grad` !

Bengio & al. (1994)

Learning Long-Term Dependencies with Gradient Descent is Difficult

Yoshua Bengio, Patrice Simard, and Paolo Frasconi, *Student Member, IEEE*

***Abstract*— Recurrent neural networks can be used to map input sequences to output sequences, such as for recognition, production or prediction problems. However, practical difficulties have been reported in training recurrent neural networks to perform tasks in which the temporal contingencies present in the input/output sequences span long intervals. We show why gradient based learning algorithms face an increasingly difficult problem as the duration of the dependencies to be captured increases. These results expose a trade-off between efficient learning by gradient descent and latching on information for long periods. Based on an understanding of this problem, alternatives to standard gradient descent are considered.**

I. Introduction

WE ARE INTERESTED IN training recurrent neural networks to map input sequences to output sequences, for applications in sequence recognition, production, or time-series prediction. All of the above applications require a system that will store and update context information; i.e., information computed from the past inputs and useful to produce desired outputs. Recurrent neural networks are well suited for those tasks because they have an internal state that can represent context information. The cycles in the graph of a recurrent network allow it to keep information about past inputs for an amount of time that is not fixed *a priori*, but rather depends on its weights and on the input data. In contrast, static networks (i.e., with no recurrent connection), even if they include delays (such as time delay neural networks [15]), have a finite impulse response and can't store a bit of information for an indefinite time. A recurrent network whose inputs are not fixed but rather constitute an input sequence can be used to transform an input sequence into an output sequence while taking into account contextual information in a flexible way. We restrict our attention here to discrete-time systems.

Learning algorithms used for recurrent networks are usually based on computing the gradient of a cost function with respect to the weights of the network [22], [21]. For example, the back-propagation through time algorithm [22] is a generalization of back-propagation for static networks in which one stores the activations of the units while going forward in time. The backward phase is also backward in time and recursively uses these activations to compute the required gradients. Other algorithms, such as the forward propagation algorithms [14], [23], are much more computationally expensive (for a fully connected recurrent network) but are local in time; i.e., they can be applied in an on-line fashion, producing a partial gradient after each time step. Another algorithm was proposed [10], [18] for training constrained recurrent networks in which dynamic neurons—with a single feedback to themselves—have only incoming connections from the input layer. It is local in time like the forward propagation algorithms and it requires computation only proportional to the number of weights, like the back-propagation through time algorithm. Unfortunately, the networks it can deal with have limited storage capabilities for dealing with general sequences [7], thus limiting their representational power.

A task displays long-term dependencies if prediction of the desired output at time t depends on input presented at an earlier time $\tau \ll t$. Although recurrent networks can in many instances outperform static networks [4], they appear more difficult to train optimally. Earlier experiments indicated that their parameters settle in sub-optimal solutions that take into account short-term dependencies but not long-term dependencies [5]. Similar results were obtained by Mozer [19]. It was found that back-propagation was not sufficiently powerful to discover contingencies spanning long temporal intervals. In this paper, we present experimental and theoretical results in order to further the understanding of this problem.

For comparison and evaluation purposes, we now list three basic requirements for a parametric dynamical system that can learn to store relevant state information. We require the following:

1) That the system be able to store information for an arbitrary duration.
2) That the system be resistant to noise (i.e., fluctuations of the inputs that are random or irrelevant to predicting a correct output).
3) That the system parameters be trainable (in reasonable time).

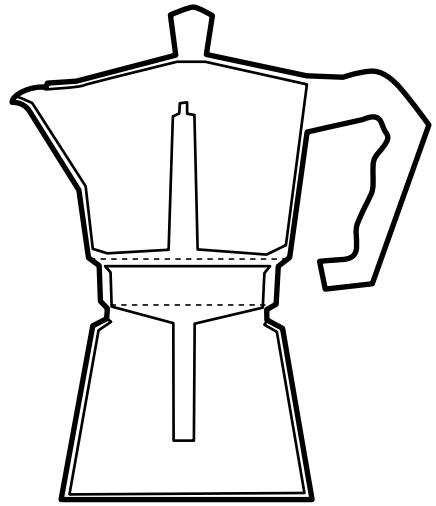
Throughout this paper, the long-term storage of definite bits of information into the state variables of the dynamic system is referred to as *information latching*. A formalization of this concept, based on hyperbolic attractors, is given in Section IV-A.

The paper is divided into five sections. In Section II we present a minimal task that can be solved only if the system satisfies the above conditions. We then present a recurrent network candidate solution and negative experimental results indicating that gradient descent is not appropriate even for such a simple problem. The theoretical results of Section IV show that either such a system is stable and resistant to noise

Manuscript received April 21, 1993; revised December 21, 1993.
Y. Bengio is with Université de Montréal (Dept. IRO), Montreal, Canada, and with AT&T Bell Laboratories, NJ.
P. Simard is with AT&T Bell Laboratories, Holmdel, NJ.
P. Frasconi is with Università di Firenze (Dip. Sistemi e Informatica), Italy.
IEEE Log Number 9215775.

We all want to work with long-term dependencies !

Temporal abstraction



Higher level steps

Choosing the type of coffee maker, type of coffee beans

Medium level steps

Grind the beans, measure the right quantity of water, boil the water

Lower level steps

Wrist and arm movements while adding coffee to the filter, ...

Temporal abstraction in AI

A cornerstone of AI planning since the 1970's:

- Fikes et al. (1972), Newell (1972, Kuipers (1979), Korf (1985), Laird (1986), Iba (1989), Drescher (1991) etc.

It has been shown to :

- Generate shorter plans
- Reduce the complexity of choosing actions
- Provide robustness against model misspecification
- Improve exploration by taking shortcuts in the environment

Frameworks for temporal abstraction in RL: MAXQ

Journal of Artificial Intelligence Research 13 (2000) 227-303 Submitted 11/99; published 11/00

Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition

Thomas G. Dietterich TGD@CS.ORST.EDU
Department of Computer Science, Oregon State University
Corvallis, OR 97331

Abstract

This paper presents a new approach to hierarchical reinforcement learning based on decomposing the target Markov decision process (MDP) into a hierarchy of smaller MDPs and decomposing the value function of the target MDP into an additive combination of the value functions of the smaller MDPs. The decomposition, known as the MAXQ decomposition, has both a procedural semantics—as a subroutine hierarchy—and a declarative semantics—as a representation of the value function of a hierarchical policy. MAXQ unifies and extends previous work on hierarchical reinforcement learning by Singh, Kaelbling, and Dayan and Hinton. It is based on the assumption that the programmer can identify useful subgoals and define subtasks that achieve these subgoals. By defining such subgoals, the programmer constrains the set of policies that need to be considered during reinforcement learning. The MAXQ value function decomposition can represent the value function of any policy that is consistent with the given hierarchy. The decomposition also creates opportunities to exploit state abstractions, so that individual MDPs within the hierarchy can ignore large parts of the state space. This is important for the practical application of the method. This paper defines the MAXQ hierarchy, proves formal results on its representational power, and establishes five conditions for the safe use of state abstractions. The paper presents an online model-free learning algorithm, MAXQ-Q, and proves that it converges with probability 1 to a kind of locally-optimal policy known as a recursively optimal policy, even in the presence of the five kinds of state abstraction. The paper evaluates the MAXQ representation and MAXQ-Q through a series of experiments in three domains and shows experimentally that MAXQ-Q (with state abstractions) converges to a recursively optimal policy much faster than flat Q learning. The fact that MAXQ learns a representation of the value function has an important benefit: it makes it possible to compute and execute an improved, non-hierarchical policy via a procedure similar to the policy improvement step of policy iteration. The paper demonstrates the effectiveness of this non-hierarchical execution experimentally. Finally, the paper concludes with a comparison to related work and a discussion of the design tradeoffs in hierarchical reinforcement learning.

Frameworks in RL: HAM

Reinforcement Learning with Hierarchies of Machines

Ron Parr and Stuart Russell
Computer Science Division, UC Berkeley, CA 94720
{parr,russell}@cs.berkeley.edu

Abstract

We present a new approach to reinforcement learning in which the policies considered by the learning process are constrained by hierarchies of partially specified machines. This allows for the use of prior knowledge to reduce the search space and provides a framework in which knowledge can be transferred across problems and in which component solutions can be recombined to solve larger and more complicated problems. Our approach can be seen as providing a link between reinforcement learning and "behavior-based" or "teleo-reactive" approaches to control. We present provably convergent algorithms for problem-solving and learning with hierarchical machines and demonstrate their effectiveness on a problem with several thousand states.

Category: reinforcement learning. **Preference**: plenary.

1 Introduction

Optimal decision making in virtually all spheres of human activity is rendered intractable by the complexity of the task environment. Generally speaking, the only way around intractability has been to provide a hierarchical organization for complex activities. Although it can yield suboptimal policies, top-down hierarchical control often reduces the complexity of decision making from exponential to linear in the size of the problem. For example, hierarchical task network (HTN) planners can generate solutions containing tens of thousands of steps [4], whereas "flat" planners can manage only tens of steps.

HTN planners are successful because they use a plan library that describes the decomposition of high-level activities into lower-level activities. This paper describes an approach to learning and decision making in *uncertain* environments (Markov decision processes) that uses a roughly analogous form of prior knowledge. We use *hierarchical abstract machines* (HAMs), which impose constraints on the policies considered by our learning algorithms. HAMs consist of nondeterministic finite state machines whose transitions may invoke lower-level machines. Nondeterminism is represented by *choice states* where the optimal action is yet to be decided or learned. The language allows a variety of prior constraints to be expressed, ranging from no constraint all the way to a fully specified solution. One useful intermediate point is the specification of just the general organization of behavior into a layered hierarchy, leaving it up to the learning algorithm to discover exactly which lower-level activities should be invoked by higher levels at each point.

The paper begins with a brief review of Markov decision processes (MDPs) and a description of hierarchical abstract

Frameworks in RL: Options

ELSEVIER

Artificial Intelligence 112 (1999) 181–211

Artificial Intelligence

www.elsevier.com/locate/artint

Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning

Richard S. Sutton [a,*], Doina Precup [b], Satinder Singh [a]

[a] *AT&T Labs.-Research, 180 Park Avenue, Florham Park, NJ 07932, USA*

[b] *Computer Science Department, University of Massachusetts, Amherst, MA 01003, USA*

Received 1 December 1998

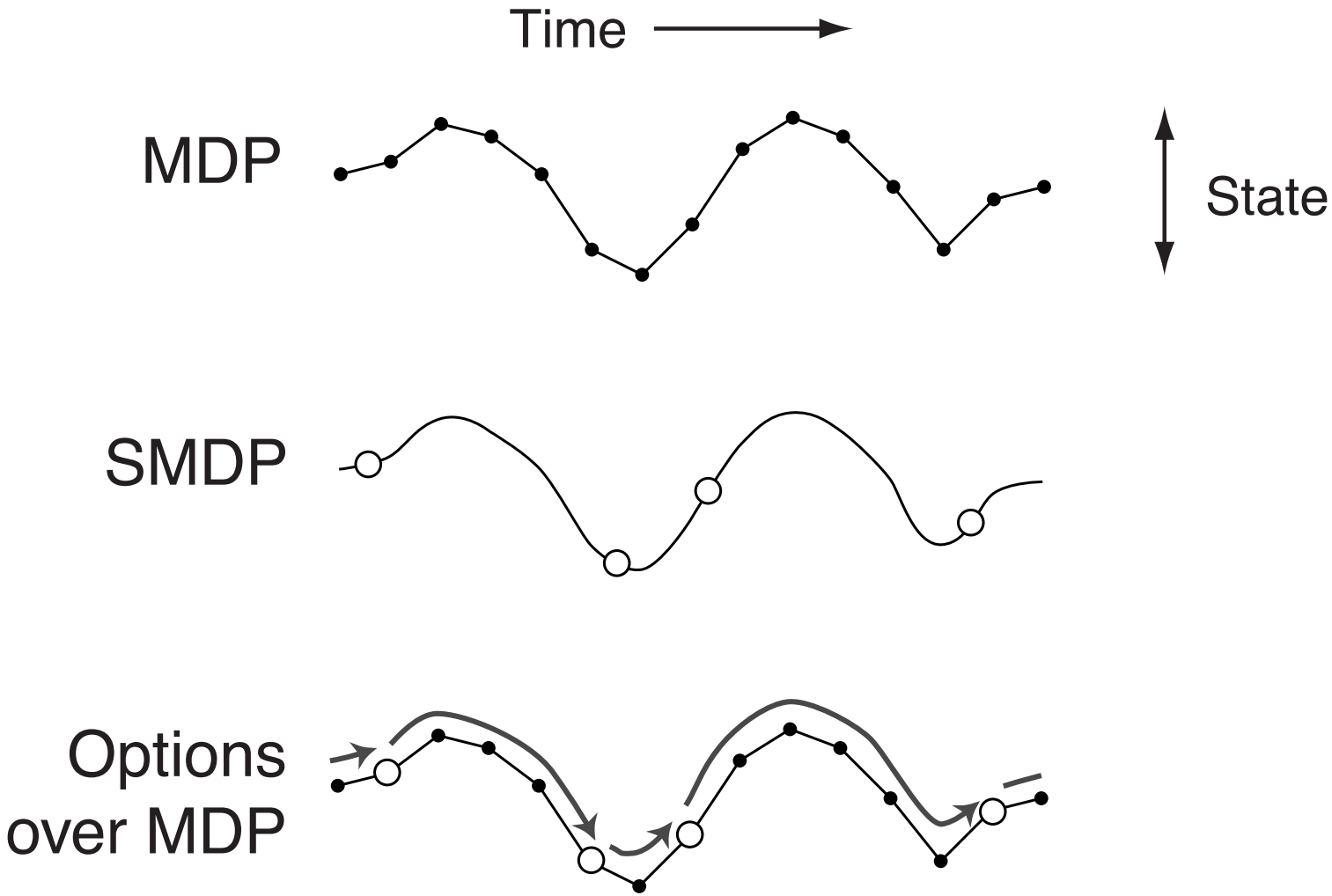
Abstract

Learning, planning, and representing knowledge at multiple levels of temporal abstraction are key, longstanding challenges for AI. In this paper we consider how these challenges can be addressed within the mathematical framework of reinforcement learning and Markov decision processes (MDPs). We extend the usual notion of action in this framework to include *options*—closed-loop policies for taking action over a period of time. Examples of options include picking up an object, going to lunch, and traveling to a distant city, as well as primitive actions such as muscle twitches and joint torques. Overall, we show that options enable temporally abstract knowledge and action to be included in the reinforcement learning framework in a natural and general way. In particular, we show that options may be used interchangeably with primitive actions in planning methods such as dynamic programming and in learning methods such as Q-learning. Formally, a set of options defined over an MDP constitutes a semi-Markov decision process (SMDP), and the theory of SMDPs provides the foundation for the theory of options. However, the most interesting issues concern the interplay between the underlying MDP and the SMDP and are thus beyond SMDP theory. We present results for three such cases: (1) we show that the results of planning with options can be used during execution to interrupt options and thereby perform even better than planned, (2) we introduce new *intra-option* methods that are able to learn about an option from fragments of its execution, and (3) we propose a notion of subgoal that can be used to improve the options themselves. All of these results have precursors in the existing literature; the contribution of this paper is to establish them in a simpler and more general setting with fewer changes to the existing reinforcement learning framework. In particular, we show that these results can be obtained without committing to (or ruling out) any particular approach to state abstraction, hierarchy, function approximation, or the macro-utility problem.

* Corresponding author.

0004-3702/99/$ – see front matter
PII: S0004-3702(99)00052-1

The "Between" in "Between MDPs and semi-MDPs"



Semi-Markov Decision Processes

In the previous slide

1. Evolution of the process over the open circles: **SMDP level** (RL terminology) or the **embedded Markov decision process** (OR terminology)
2. Evolution of the process with open circles removed: **flat level** (RL terminology), also called **natural process** (OR terminology)

Trajectories in an SMDP

$$\tau \in \{\mathcal{T} \times \mathcal{S} \times \mathcal{A}\}^{\infty}$$

$$(S_0, A_0, T_0, S_1, A_1, T_1, \ldots)$$

We now have a distribution of **transition times** between two decisions.

Options framework

A Markov option ω is a triple:

$$\Big(\underbrace{\mathcal{I}_w \subseteq \mathcal{S}}_{\text{initiation set}} \;,\; \underbrace{\pi_w : \mathcal{S} \times \mathcal{A} \to [0,1]}_{\text{internal policy}},\; \underbrace{\beta_w : \mathcal{S} \to [0,1]}_{\text{termination function}} \Big)$$

and we also define a **policy over options** $\mu : \mathcal{S} \times \mathcal{W} \to [0,1]$.

Example

Robot navigation: if there is no obstacle in front ($\mathcal{I}_w$), go forward (π_w) until you get too close to another object (β_w)

Option models

Given a set of options and an MDP, we can define **option models**

1. **Expected reward** $b_w(s)$: the expected return during w's execution from s
2. **Transition model** $F_w(s', s)$: a distribution over next states (reflecting the discount factor γ and the option duration) given that w executes from s
   - F_w specifies *where* the agent will end up after the option/program execution and *when* termination will happen

Models are *predictions* about the future, conditioned on the option being executed.

Bellman equations for options

At the SMDP level, everything behaves like an MDP over *transformed* reward and transition functions/models:

$$Q_\mu(s,w) = b_w(s) + \sum_{s'} F_w(s,s') v_\mu(s')$$

$$v_\mu(s) = \sum_w \mu(w \mid s)\, Q_\mu(s,w)$$

Optimality Equations:

$$v^*(s) = \max_w \left(b_w(s) + \sum_{s'} F_w(s,s') v^*(s') \right)$$

Usual DP methods for MDPs directly apply ! Same for Dyna, or TD.

TD at the SMDP level

Let N_t now be a random variable for the duration of an option started at time t:

$$G_t^{(N_t)} = R_t + \gamma R_{t+1} + \ldots + \gamma^{N_t-1} R_{t+N_t-1} + \gamma^{N_t} Q(S_{t+N_t}, W_{t+N_t})$$

Key idea

It's just like using the n-steps return but where n is a random variable rather than being fixed.

SMDP TD Prediction (for option values)

$$Q_{t+N_t}(S_t, W_t) = Q_{t+N_t-1}(S_t, W_t) + \alpha \left(G_t^{(N_t)} - Q_{t+N_t-1}(S_t, W_t) \right)$$

Question : how do you get SMDP Q-learning ? **Answer**: just take a $\max_w$ in the $G_t^{(N_t)}$ update target.