We are IntechOpen, the world’s leading publisher of Open Access books
Built by scientists, for scientists

4,300
Open access books available

116,000
International authors and editors

125M
Downloads

154
Countries delivered to

TOP 1%
Our authors are among the most cited scientists

12.2%
Contributors from top 500 universities

WEB OF SCIENCE™
Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com
1. Introduction

Neuroscientists believe that living beings solve the daily life activities, making decisions and hence adapt to newly situations by learning from past experiences. Learning from experience implies that each event is learnt through features (i.e. sensory control inputs) analysis that aimed to specify and then recall more important features for each event or situation.

In robot learning, several works seem to suggest that the transition to the current reinforcement learning (RL) (1), as a general formalism, does correspond to observable mammal brain functionality, where ‘basal ganglia’ can be modeled by an actor-critic (AC) version of temporal difference (TD) learning (2; 3; 4). However, as with the most real-world intelligent learning systems, the arising of ‘perceptual aliasing’ (also referred to as a problem of ‘incomplete perception’, or ‘hidden state’) (5), when the system has to scale up to deal with complex nonlinear search spaces in a non-Markov settings or Partially Observation Markov Decision Process (POMDP) domains (6) (see Fig. 1) renders to-date RL methods impracticable, and that they must learn to estimate value function \( v^\pi \) instead of learning the policy \( \pi \); limiting them mostly for solving only simple learning tasks, raising an interest in heuristic methods that directly and adaptively modifying the learning policy \( \pi: S \rightarrow A \) (which maps perceptual state/observation to action) via interaction with the rest of the system (7; 8).

Inclusion of a memory to a simulated robot control system is striking because a memory learning system has the advantage to deal with perceptual aliasing in POMDP, where memoryless policies are often fail to converge (9).

In this paper, a self-optimizing memory controller is designed particularly for solving non-Markovian tasks, which correspond to a great deal of real-life stochastic predictions and control problems (10) (Fig. 2). Rather than holistic search for the whole memory contents the controller adopts associated feature analysis to successively memorize a newly experience (state-action pair) as an action of past experience. e.g., If each past experience was a chunk, the controller finds the best chunk for the current situation for policy exploration. Our aim is not to mimic the neuroanatomical structure of the brain system but to catch its properties, avoids manual ‘hard coding’ of behaviors. AC learning is used to adaptively tune the control parameters, while an on-line variant of decision-tree ensemble learner (11; 12) is used as memory-capable function approximator coupled with Intrinsically Motivated Reinforcement Learning (IMRL) reward function (13; 14; 15; 16) to approximate the policy of
Fig. 1. POMDP and Perceptual aliasing. RL agent is connected to its world via perception state $S$ and action $A$. In (a) a partially observable world, in which the agent does not know which state it is in due to sensor limitations; for the value function $v^\pi$, the agent updates its policy parameters directly. In (b) and (c) two maze domains. States indicated with the same letter ($X$ or $Y$) are perceptually aliased because the agent is sensed only wall configuration.

the actor and the value function of the critic. Section 2 briefly highlights on POMDP settings. A description with comprehensive illustration of the proposed memory controller will be given in Section 3. Then Section 4 highlights a comparison of conventional memory controller and the self-optimizing memory controller. Section 5 shows the implementation of decision-tree ensemble as memory-capable function approximator for both critic and policy. Some experimental results are presented in Section 6 as promising examples. It includes the non-Markovian cart-pole balancing tasks. The results show that our controller is able to memorize complete non-Markovian sequential tasks and develop complex behaviors such as balancing two poles simultaneously.

2. A non-Markovian and perceptual aliasing
First we present the formal setting of POMDP and then highlight on related approaches tackling perceptual aliasing.

2.1 POMDP formal setting
The formal setting of POMDP is $\mathcal{P} = (\mathcal{M}, \mathcal{O}, \mathcal{Z})$ consist of:

1. An MDP of a tuple $\mathcal{M}= (S,A,T,R)$ where $S$ is the space of possible states of the environment, $A$ is a set of actions available to the agent (or control input), $P : S \times A \times S \rightarrow [0,1]$ defines a conditional probability distribution over state transitions given an action, and $R : S \times A \rightarrow R$ is a reward function (payoff) assigning a reward for an action,

2. A set of possible observations $\mathcal{O}$, where $O$ could constitute either a set of discrete observations or a set of real-value,
3. \( Z \), a probability density mapping state-observation combinations \( S \times O \) to a probability distribution, or in the case of discrete observations combinations \( S \times O \) to probabilities. In other words, \( Z(s, o) \) yields the probability to observing \( o \) in state \( s \). So basically, a POMDP is like an MDP but with observations instead of direct state perception.

If a world model is available to the controller, it can easily calculate and update a belief vector \( \hat{b}_t = \left( \hat{b}_t(s_1), \hat{b}_t(s_2), \ldots, \hat{b}_t(s_N) \right) \) over 'hidden states' at every time step \( t \) by taking into account the history trace \( h = o_1, o_2, \ldots, o_{t-1}, o_t \).

2.2 Perceptual aliasing

It is important to note that in several literatures, perceptual aliasing is wrongly defined as the problem of having an incomplete instance, whereas this paper defines it as a problem related to having different states that may look similar but are related to different responses. Uncomplete instances may provoke perceptual aliasing, but they are not the same. Although the solely work in this paper is focused on POMDP, we briefly highlight on related approaches, in order to decipher the ambiguities between POMDP and perceptual aliasing:

- **Hidden Markov Models** (HMMs): are indeed applied to the more general problem of perceptual aliasing. In HMM it is accepted that we do not have control over the state transitions, whereas POMDP assume that we do. Hence, POMDP are more related to incomplete perception than to perceptual aliasing. HMMs have been thoroughly applied to robotic behavior synthesis, see, for example (18).

- **Memory-based system**: in Memory-based systems the controller is unable to take optimal transitions unless it observed the past inputs, then the controller simultaneously solve the incomplete perception while maximizing discounted long-term reward. For an early practice attempts with other alternative POMDP approaches, e.g., the ‘model-based approach or belief-based approach’, and the ‘heuristic method with a world model’ within TD reinforcement learning domain, see (23; 24).

- There is a large body of work on behavior learning both supervisedly and unsupervisedly using fuzzy logic, Artificial Neural Networks (ANN) and/or Case Based Reasoning (CBR). Some of them do not establish rules and, specifically, CBR uses memory as its key learning tool. This, too, has been used in robotics in loosely defined navigation problems. See, for example (19).

3. Self-optimizing controller architecture

One departing approach from manual ‘hard coding’ of behaviors is to let the controller build its own internal ‘behavior model’ ‘on-the-fly’ by learning from past experience. Fig. 2 illustrates the general view of our memory controller based on heuristic memory approach. We briefly explain its components. It is worth noted that in our implementation only the the capacity of the memory and reward function have be specified by a designer, the controller is self-optimized in a sense that we do not analyzing a domain a priori, instead we add an initially suboptimal model, which is optimized through learning\(^1\).

---

\(^1\) At this point we would like to mention that M3 Computer Architecture Group at Cornell has proposed a similar work (17) to our current interest. They implement a RL-based memory controller with a different underlying RL implementation, we inspired by them in some parts.
Past experiences. Sensory control inputs from environment would be stored at the next available empty memory location (chunk), or randomly at several empty locations.

**Feature predictor.** Is utilized to produce associated features for each selective experience. This predictor was designed to predict multiple experiences in different situations. When the selective experience is predicted, the associated features are converted to feature vector so the controller can handle it.

**Features Map.** The past experiences are mapped into multidimensional feature space using neighborhood component analysis (NCA) \((20; 21)\), based on the Bellman error, or on the temporal difference (TD) error. In general this is done by choosing a set of features which approximate the states \(S\) of the system. A function approximator (FA) must map these features into \(V^{π}\) for each state in the system. This generalizes learning over similar states and more likely to increase learning speed, but potentially introduces generalization error as the feature will not represent the state space exactly.

**Memory access.** The memory access scheduling is formulated as a RL agent whose goal is to learn automatically an optimal memory scheduling policy via interaction with the rest of the system. A similar architecture that exploits heterogeneous learning modules simultaneously has been proposed \((22)\). As can be seen in the middle of Fig. 2 two scenarios are considered. In (a) all the system parameters are fully observable, the agent can estimate \(v^{π}\) for each state and use its actions (e.g., past experiences). The agent’s behavior, \(B\), takes actions that tend to increase the long-run sum of values of the reinforcement signal, typically \([0,1]\). In (b) the system is partially observable as described in Fig. 1. Since our system is modeled as POMDP decision depends on last observation-action, and the observation transitions \(s_{t+1} = \delta(s_t, a_t)\) depend on randomly past perceptual state. This transition is expressed by \(Pr(s_t | s_{t-1}, a_{t-1}, s_t', s_t', ..)\), where \(s_{t-1}, a_{t-1}\) are the previous state and action, and \(t',t''\) are arbitrary past time.

**Learning behaviors from past experience.** On each time step \(t\), an adaptive critic (that is a component of the TD learning), is used to estimate future values of the reinforcement signal of retaining different memory locations, which represents the agent’s behavior, \(B\) in choosing actions. The combinations of memory locations show to have the highest accumulated signals are more likely to be remembered. TD error—the change in expected future signal is computed based on the amount of occasional intrinsic reinforcement signal received, a long with the estimates of the adaptive critic.

4. Non-Markovian memory controller

4.1 Conventional memory controller

Conventional manually designed memory controller suffers two major limitations in regard with scheduling process and generalization capacity. First, it can not anticipate the long-term planning of its scheduling decisions. Second, it lacks learning ability, as it can not generalize and use the experience obtained through scheduling decisions made in the past to act successfully in new system states. This rigidity and lack of adaptivity can lead to severe performance degradation in many applications, raising interest in self-optimizing memory controller with generalization capacity.

4.2 Self-optimizing memory controller

The proposed self-optimizing memory controller is a fully-parallel maximum-likelihood search engine for recalling the most relevant features in the memory of past. The memory
controller considers the long-term planning of each available action. Unlike conventional memory controllers, self-optimizing memory controller has the following capabilities: 1) Utilizes experience learnt in previous system states to make good scheduling decisions in new, previously unobserved states, 2) Adapts to the time-variant system in which the state transition function (or probability) is permitted to gradually change through time, and 3) Anticipates the long-term consequences of its scheduling decisions, and continuously optimizes its scheduling policy based on this anticipation.

No key words or pre-determined specified memory locations would be given for the stored experiences. Rather a parallel search for the memory contents would take place to recall the previously stored experience that correlates with the current newly experience. The controller handle the following tasks: (1) relate states and actions with the occasional reward for long planning, (2) take the action that is estimated to provide the highest reward value at a given state, and (3) continuously update long-term reward values associated with state-action pairs, based on IMRL.

www.intechopen.com
5. Memory-capable function approximation

5.1 Actor-critic learning

Actor-critic (AC), a group of on-policy TD methods, separates the $\pi$ and the $v^\pi$ into independent memory structures. The $\pi$ structure, or actor, is used to decide which action to pick in each state. The estimate of $v^\pi$, or adaptive critic, determines whether the actions of the actor are to be rewarded or punished. The algorithms use these spare measures of performance to adopt an optimal behavior over time. The adaptive critic maps its current state event onto an estimate of whether it will be rewarded. The mapping is learned from the past experience. If $s + 1$ is the situation that follows situation $s$ in time, this expected future reward may be written as:

$$V(s) = \gamma^0 r(s) + \gamma^1 V(s+1) + \cdots + \gamma^n V(s+n)$$  \hspace{1cm} (1)

The value of the current situation, $V(s)$, is the sum of all the rewards we will receive over the next $n$ time steps. The rewards on each time step are “discounted” by factor, $\gamma$, in the range $[0,1]$. Equation (1) can be rewritten in a recursive form:

$$V(s) = \gamma^0 r(s) + \gamma^1 V(s+1) = r(s) + \gamma V(s+1)$$  \hspace{1cm} (2)

It should be noted that the equality in Eq. 2 is valid only if $n$ is infinite or the state at $n$ time steps later, $s + n$, is always a so-called ‘absorbing state.’ Obviously a value function estimates that fall far from this equality in considered inaccurate, and the error is estimated based on TD error:

$$\delta(s) = (r(s) + \gamma V(s+1) - V(s))$$ \hspace{1cm} (3)

Adopting these methods can save much computation for selecting optimal actions, due to utilizing separate memory for value function and policy.

5.2 AC in non-Markovian domain

Due to non-Markovian characteristics, the controller infers the state of its environment from a sequence of observations it receives, learns an optimal action by detecting certain past events, that associated with its current perception. In particular, at time $t$, the error of the critic is given by,

$$E_c(t) = \frac{1}{2} [(r(t) + \gamma J(t))] - J(t-1))^2 \hspace{1cm} (4)$$

while the error of the actor is

$$E_a(t) = \frac{1}{2} (J(t) - R^*)^2 \hspace{1cm} (5)$$

where $R^*$ is the optimal return, which is dependent on the problem definition. The expected return is expressed as the general utility function, $J(t)$, which is to be maximized by the controller. Specifically,

$$J(t) = r(t+1) + \gamma r(t+2) + \gamma^2 r(t+3) + \cdots \hspace{1cm} (6)$$

where $r(t)$ is the immediate reward and $\gamma$ is the time-discounting factor $0 \leq \gamma \leq 1$. 

www.intechopen.com
5.3 Decision-tree ensemble memory for optimal learning
On-line decision-tree ensemble learner has the characteristics of a simple structure, strong
global approximation ability and a quick and easy training (11; 12). It has been used with TD
learning for building a hybrid function approximator (26; 27). Here, in order to improve
learning efficiency and to reduce the demand of storage space and to improve learning
efficiency, the on-line ecision-tree ensemble approximator is structured in a way that both
actor and critic can be embodied in one structure, subsequently, is used to approximate π
of the actor and the vπ of the critic simultaneously. That is, the actor and the critic can share the
input and the basis functions structure of the RF. Let DTAppro represents a hybrid
approximator that combines actor and critic. Given a state s(t) and action a(t), DTAppro is
defined such that DTAppro(s(t), a(t)) = (J(t), a(t+1)), where J(t) is the estimated value of the
given state-action pair, and a(t+1) is the subsequent action to be taken by the controller. At
the critic output the error is captured by TD error. However, at the action outputs the error
is determined by the gradient of the estimated value J(t + 1) w.r.t the action a(t + 1) selected
by the on-line RF at time t. Specifically,
\[ e_a(t) = \alpha V_{a(t+1)}(t+1) \]
\[ = \alpha \left( \frac{\partial J(t+1)}{\partial u_1(t+1)} \ldots \frac{\partial J(t+1)}{\partial u_d(t+1)} \right) \]  
(7)
where \( \alpha \) is a scaling constant and \( d \) is the choices availabilities at action \( a \). Accumulating the
error for each choice of the selected action, the overall actor error is given be:
\[ E_a(t) = \frac{1}{2} \sum_{i=1}^{d} e^2_a(t) \]  
(8)
where \( e_a(t) \) is the choice of the action error gradient \( e_a(t) \). In finding the gradient of the
estimated value \( J(t + 1) \) w.r.t the previously selected action \( a(t + 1) \), the direction of change in
action, which will improve the expected return at time step \( t + 1 \), is obtained. Thus by
incrementally improving actions in this manner, an optimal policy can be achieved. \( E(t) \) = \( E_a(t) + E_c(t) \) defines the reduced error for the entire on-line approximator.

6. Experiment and results
As discussed in previous sections, the proposed controller brings a number of preferable
properties for learning different behaviors. In this section, we investigate its learning capability
through a task of cart-pole balancing problem, designed with non-Markovian settings.

6.1 Related work
Modeling the pole balancing algorithm for POMDP has received much interest in the field
on control and artificial intelligence. Although a variation of Value and Policy Search
(VAPS) algorithm (28) has been applied to this problem for the POMDP case (29), they have
assumed that the position of cart on track \( x \) and the angle of pole from vertical \( \theta \) are
completely observable. NeuroEvolution of Augmenting Topologies (30) and evolutionary
computation (31), are another promising approaches where recurrent neural networks are
used to solve a harder balancing of two poles of different lengths, in both Markovian and
non-Markovian settings.
6.2 Non-Markovian Cart Pole balancing

As illustrated in Fig. 3A, Cart-Pole balancing involves a vertical pole with a point-mass at its upper end installed on a cart, with the goal of balancing the pole when the cart moves by applying horizontal forces to the cart, which must not stray too far from its initial position. The state description for the controller consists of four continuous state variables, the angle \( \theta \) (radial), and the speed of the pole \( \dot{\theta} = \delta x / \delta t \) plus the position \( x \) and speed of the cart \( \dot{x} = \delta x / \delta t \), (see Appendix A.1 for the equations of motion and Appendix A.2 for parameters used as reported by (31)). The two continuous actions set up for controller training and evaluation were RightForce (RF), (results in pushing the cart to the right), and LeftForce (LF), (results in pushing the cart left). At each time step \( t \), the controller must only observe the \( \theta \) (that is, the controller is not observing the velocities \( \dot{x}, \dot{\theta} \)) and then takes appropriate action to balance the pole by learning from the past experience and the intrinsically rewards. The optimal value function is shown in Fig. 3B. A simulated sample run is shown in Fig. 4. The controller could keep the pole balanced after about 4000 steps.

Fig. 3. (A) Illustration of the non-Markov Cart-Pole balancing problem, where the angular velocity is not observing by the controller. (B) Optimal value function.

Fig. 4. A sample learning for balancing the pole. It suggests that the method could keep the pole near the top for a long time.
6.3 Non-Markovian Two-Pole balancing

Then we moved to a harder setting of this problem, balancing two poles simultaneously (see Fig. 5). Each pole has its own position and angular velocity, $\theta_1$ and $\dot{\theta}_1$ for the first pole and $\theta_2$ and $\dot{\theta}_2$ for the second pole respectively. See Appendix A.2 for parameters used as reported by (31). The controller must balance the two poles without velocity information. In order to assist the feasibility of our approach to balance two poles simultaneously we compared with other methods.

Fig. 5. Illustration of the non-Markov 2-Pole balancing problem. Parameters known are $\theta_1$ and $\theta_2$. The controller must balance the two poles without observing $\dot{\theta}_1$ and $\dot{\theta}_2$.

Table 1 reports the performance of our controller compared with traditional value function based methods (See Appendix B.1 for parameters used) (including SARSA-CABA (See Appendix B.2, SARSA-CMAC (See Appendix B.3, which are reported by (31), who used SARSA implementations by (32) and VAPS (See Appendix B.4) and policy search method (including Q-MLP (See Appendix B.5, as implementation of (31)). Table 1 shows that our controller takes the minimal evaluations to balance the poles. With regard to CPU time (reported in seconds) we slightly fall short to Q-MLP. However, it interesting to observe that none of the value function approaches could handle this task in within the set of steps (e.g., 100,000 time steps, which is equal to over 30 minutes in simulated time) due to the memory constraint. The result also indicates that our memory controller stand as a promising method in solving this benchmark more successful than the traditional RL techniques.

<table>
<thead>
<tr>
<th>Method</th>
<th>Evaluation</th>
<th>time (second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>V-function</td>
<td>SARSA-CMAC</td>
<td>Time Out</td>
</tr>
<tr>
<td></td>
<td>SARSA-CABA</td>
<td>Time Out</td>
</tr>
<tr>
<td></td>
<td>VAPS</td>
<td>Time Out</td>
</tr>
<tr>
<td>Policy</td>
<td>Q-MLP</td>
<td>10,582</td>
</tr>
<tr>
<td>Memory</td>
<td>Our</td>
<td>8,900</td>
</tr>
</tbody>
</table>

Table 1. Comparison of our result for balancing two-pole simultaneously with other value function approaches and policy based methods. ‘Evaluation’ indicates the total time steps for the method to be able to keep the poles near the top for a long time.

7. Conclusions

This paper proposes an architecture which avoids manual ‘hard coding’ of behaviors, where an RL agent uses an adaptive memory process to create its own memory and thereby
perform better in partially observable domains. The algorithm uses neighborhood component analysis (NCA) to determine feature vectors for system states. Decision-trees ensemble is used to create features which are useful in predicting the state of the system (i.e. building some sort of forward model). Chunks are used with a feature predictor to get features. These features are then used as the input features to learn a policy. Results based on non-Markov Cart-Pole balancing indicate that our model can memorize complete non-Markovian sequential tasks and is able to produce behaviors that make the controlled system to behave desirably in the future. One of our future plans is to automate the capacity of memory in order to accommodate more complex tasks. In our current design the number of chunks that can be used is fixed. Another future plan will be in designing intelligent mechanism for memory updating, and to experiment with real world applications.

8. References


**APPENDIX**

**A. Pole-balancing learning parameters**

Below are the equations and parameters used for cart-pole balancing experiments (31)

**A.1 Pole-balancing equations**

The equations of motion for *N* unjoined poles balanced on a single cart are

\[
\ddot{x} = \frac{F - \mu \text{sgn}(\dot{x}) + \sum_{i=1}^{N} \ddot{F}_i}{M + \sum_{i=1}^{N} \ddot{m}_i}
\]

\[
\dot{\theta} = -\frac{3}{4l} (\ddot{x} \cos \theta + g \sin \theta + \frac{\mu \dot{\theta}}{m_i l_i})
\]

where \( \ddot{F}_i \) is the effective force from the \( i \)th pole on the cart,

\[
\ddot{F}_i = m_i l_i \dot{\theta}_i^2 \sin \theta_i + \frac{3}{4} m_i \cos \theta_i \left( \frac{\mu \dot{\theta}_i}{m_i l_i} + g \sin \theta_i \right),
\]

and \( \ddot{m}_i \) is the effective mass of the \( i \)th pole,

\[
\ddot{m}_i = m_i (1 - \frac{3}{4} \cos^2 \theta_i).
\]
A.2 Pole-balancing learning parameters

<table>
<thead>
<tr>
<th>Parameters for the single pole</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Sym</td>
<td>Description</td>
<td>Value</td>
</tr>
<tr>
<td>$x$</td>
<td>Position of cart on track</td>
<td>$[-2.4, 2.4]$ m</td>
</tr>
<tr>
<td>$\theta$</td>
<td>Angle of pole from vertical</td>
<td>$[-12, 12]$ deg</td>
</tr>
<tr>
<td>$F$</td>
<td>Force applied to cart</td>
<td>$-10.10$ N</td>
</tr>
<tr>
<td>$l$</td>
<td>Half length of pole</td>
<td>$0.5$ m</td>
</tr>
<tr>
<td>$m_c$</td>
<td>Mass of cart</td>
<td>$1.0$ kg</td>
</tr>
<tr>
<td>$m_p$</td>
<td>Mass of pole</td>
<td>$0.1$ kg</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameters for double pole</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Sym</td>
<td>Description</td>
<td>Value</td>
</tr>
<tr>
<td>$x$</td>
<td>Position of cart on track</td>
<td>$[-2.4, 2.4]$ m</td>
</tr>
<tr>
<td>$\theta$</td>
<td>Angle of pole from vertical</td>
<td>$[-36, 36]$ deg</td>
</tr>
<tr>
<td>$F$</td>
<td>Force applied to cart</td>
<td>$-10.10$ N</td>
</tr>
<tr>
<td>$l_i$</td>
<td>Half length of $i^{th}$ pole</td>
<td>$l_1 = 0.5$ m, $l_2 = 0.05$ m</td>
</tr>
<tr>
<td>$m_c$</td>
<td>Mass of cart</td>
<td>$1.0$ kg</td>
</tr>
<tr>
<td>$m_{p_i}$</td>
<td>Mass of $i^{th}$ pole</td>
<td>$m_{p_1} = 0.1$ kg, $m_{p_2} = 0.01$ kg</td>
</tr>
<tr>
<td>$\mu_c$</td>
<td>Friction coef on cart on track</td>
<td>$0.0005$</td>
</tr>
<tr>
<td>$\mu_p$</td>
<td>Friction coef if $i^{th}$ pole’s hinge</td>
<td>$0.0005$</td>
</tr>
</tbody>
</table>

Table 2. Parameters for the single pole & double pole problem.

B. Parameters for comparisons in cart pole balancing

Below are the parameters used to obtain the comparison result for SARSA-CABA, SARSA-CMAC, Q-MLP (31), and VAPS (28) in Section 6.3.

B.1 Parameters for value function methods

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
<td>Greediness of policy</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>Learning rate</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>Discount rate</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>Eligibility</td>
</tr>
</tbody>
</table>

Table 3. Parameters for value function methods.

B.2 Parameters used for SARSA-CABA

Sarsa($\lambda$) with Case-Based function approximator (SARSA-CABA (32)): Is a Sarsa method with $\lambda$ that uses a case-based memory to approximate the Q-function. A newly added state-action pair is calculated by combining the values of the k-nearest neighbors.

B.3 Parameters used for SARSA-CMAC

Sarsa($\lambda$) with CMAC function approximator (SARSA-CMAC (32)): Almost similar to SARSA-CABA except that it uses a Cerebellar Model Articulation Controller (CMAC) instead of a case-based memory to approximate the Q-function. Using this method the state-action space is divided into a set of tilings, each tiling constitutes a set of discrete features. Q-value is calculated as the sum of the value in each tiling.
Table 4. Parameters used for SARSA-CABA.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma_d$</td>
<td>0.03</td>
</tr>
<tr>
<td>$\Gamma_k$</td>
<td>0.05</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>0.4</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>0.99</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>0.4</td>
</tr>
</tbody>
</table>

B.4 Value and policy search

(VAPS (28)): Is an extension of a method proposed by (35) to policies graph, where stochastic gradient descent is used to search the space. The graph is made of ‘nodes’ indicating actions and ‘arcs’ representing observation. Transitions between nodes are initially based on the action associated with node that the agent previously visited, while the environment continue to produce arcs labeled with observations.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
<td>0.05</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>0.4</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>0.9</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Table 5. Parameters used for SARSA-CMAC.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
<td>0.1</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>0.4</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>0.9</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>0</td>
</tr>
</tbody>
</table>

B.5 Parameters used for Q-MLP

Q-learning with MLP (Q-MLP): This method uses a Multi-Layer Perceptron to map state-action pairs to $Q(s, a)$ that makes it different from standard Q-learning (36). Backpropagation algorithm is used to learn the network values through gradient descent, produces a single Q-value as the output layer. This approach has been thoroughly applied to pole-balancing (37), and backgammon (38).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
<td>0.1</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>0.4</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>0.9</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6. Parameters used for Q-LMP.
Robot Learning is intended for one term advanced Machine Learning courses taken by students from different computer science research disciplines. This text has all the features of a renowned best selling text. It gives a focused introduction to the primary themes in a Robot learning course and demonstrates the relevance and practicality of various Machine Learning algorithms to a wide variety of real-world applications from evolutionary techniques to reinforcement learning, classification, control, uncertainty and many other important fields. Salient features: - Comprehensive coverage of Evolutionary Techniques, Reinforcement Learning and Uncertainty. - Precise mathematical language used without excessive formalism and abstraction. - Included applications demonstrate the utility of the subject in terms of real-world problems. - A separate chapter on Anticipatory-mechanisms-of-human-sensory-motor-coordination and biped locomotion. - Collection of most recent research on Robot Learning.

How to reference
In order to correctly reference this scholarly work, feel free to copy and paste the following:
