Integrated Learning and Planning Based on Truncating Temporal Differences

Paweł Cichosz

Institute of Electronics Fundamentals
Warsaw University of Technology
Nowowiejska 15/19, 00-665 Warsaw, Poland
phone: +48-22/660-77-18, fax: +48-22/25-23-00
cichosz@ipe.pw.edu.pl
http://www.ipe.pw.edu.pl/~cichosz

Abstract. Reinforcement learning systems learn to act in an uncertain environment by executing actions and observing their long-term effects. A large number of time steps may be required before this trial-and-error process converges to a satisfactory policy. It is highly desirable that the number of experiences needed by the system to learn to perform its task be minimized, particularly if making errors costs much. One approach to achieve this goal is to use hypothetical experiences, which requires some additional computation, but may reduce the necessary number of much more costly real experiences. This well-known idea of augmenting reinforcement learning by planning is revisited in this paper in the context of truncated TD(λ), or TTD, a simple computational technique which allows reinforcement learning algorithms based on the methods of temporal differences to learn considerably faster with essentially no additional computational expense. Two different ways of combining TTD with planning are proposed which make it possible to benefit from $\lambda > 0$ in both the learning and planning processes. The algorithms are evaluated experimentally on a family of grid path-finding tasks and shown to indeed yield a considerable reduction of the number of real interactions with the environment necessary to converge, as well as an improvement of scaling properties.

1 Introduction

A *reinforcement learning* (RL) system at each step of discrete time observes the current *state* of its environment and executes an *action*. Then it receives a *reinforcement*, or reward value, and a state transition takes place. Reinforcement values provide a measure of the quality of actions executed by the system. The objective of learning is to identify a *decision policy* (i.e., a state-action mapping) that maximizes the reinforcement values received by the learner *in the long term*. A typical performance measure is the expected total discounted sum of reinforcement it receives during its lifetime in continuous learning tasks, or during a trial in episodic tasks. Rewards from subsequent time steps are multiplied by the subsequent powers of a *discount factor* $0 \leq \gamma \leq 1$, which, for positive γ, makes the learner take into account not only the immediate, but also the delayed consequences of its actions, and, for $\gamma < 1$, to consider the rewards received soon more important that those from the far future. This involves the temporal credit assignment problem, commonly solved using algorithms based on the methods of *temporal differences* (TD) [11].

The key problem with the existing reinforcement learning algorithms is that they usually converge slowly, especially for tasks with large state spaces. A considerable part of current RL research is devoted to various possible ways of overcoming this painful deficiency. Of those, it is particularly worthwhile to mention state generalization by the use of function approximators [13,3], hierarchical RL architectures [7,6], and integrating learning with planning [12,9]. The last approach is investigated in this paper in a new, promising context. Novel hybrid learning and planning algorithms are proposed, obtained by combining the frameworks of the Dyna architecture [12] and of the TTD procedure [2]. These two are briefly described in the next section.

2 Background

The Dyna architecture is a generic instantiation of the idea of combining reinforcement learning with planning by means of learning from hypothetical experiences. The TTD procedure is a simple computational technique that allows one to efficiently implement TD-based reinforcement learning algorithms in their TD($\lambda > 0$) versions, which usually converge faster than the simplest TD(0) ones. The discussion of these two foundations for this work is limited to the most essential points. The reader may refer to the cited literature for more details.

2.1 Dyna Architecture

The basic idea of Sutton's Dyna framework [12] is that one can reduce the number of interactions with the "real world" necessary to learn an acceptable policy by introducing a number of hypothetical interactions with a model, which predicts the consequences of the learner's actions in different states. The model is usually not known *a priori* and thus it is learned during the regular operation of the learning system. Hypothetical experiences are processed by essentially the same reinforcement learning algorithm as real ones. Learning from experiences generated by the model is referred to as planning.

The generic Dyna algorithm is shown in Figure 1, as a sequence of operations performed at each time step. The current real experience, $\langle x_t, a_t, r_t, x_{t+1} \rangle$, is used to update the model in Step 4 and then passed to the underlying reinforcement learning algorithm, referred to as $\mathbb{RL}$. Additionally, a number of hypothetical experiences are processed, generated by applying a (forwards) model to different state-action pairs.

The operation written as $\mathbb{FM}(x, a)$ is intended to return a pair (r, x') such that executing action a in state x is predicted to result in a transition to state x' and a reinforcement value of r. Of course a model must be sufficiently accurate to

```
At each time step $t$:
1. observe current state $x_t$;
2. $a_t :=$ select_action($x_t$);
3. perform action $a_t$; observe new state $x_{t+1}$ and immediate reinforcement $r_t$;
4. update_model($x_t, a_t, r_t, x_{t+1}$);
5. $\mathbb{RL}(x_t, a_t, r_t, x_{t+1})$;
6. repeat $K$ times
    (a) choose a hypothetical state $x$;
    (b) $a :=$ select_action($x$);
    (c) $(r, x') := \mathbb{FM}(x, a)$;
    (d) $\mathbb{RL}(x, a, r, x')$.
```

Fig. 1. The generic Dyna algorithm.

be useful, and a very poor model may be harmful. The learner may be supplied with some initial model as a part of its background "innate" knowledge. Each real experience may be used to update the model and to verify its reliability. Planning should be performed only if the current model is found to be reliable—in a practical implementation of the algorithm from Figure 1, Step 6 should be probably conditioned by the reliability status of the model.

In order to instantiate the Dyna architecture, one has to select its three major components: model representation and learning algorithms, a strategy for choosing hypothetical experiences, and a reinforcement learning algorithm. The task of learning a model is a typical *supervised learning* task and there are several methods that can be applied to accomplish it. The proper choice of one of them depends, first of all, on the state and action representation. Peng and Williams [9], and independently Moore and Atkeson [8] developed heuristic planning strategies for deciding which hypothetical experiences are most promising and should be presented to the learner. The essential idea of these heuristics is to assign a priority number to each of candidate experiences, based on the magnitude of the reinforcement learning error value recently used to update some of its possible successor states, and to maintain a priority queue of experiences. Then each time when a planning step is to be performed, a promising hypothetical experience may be extracted from the queue. This simple idea is reused by one of the algorithms proposed later in this paper.

2.2 Q-Learning

The Q-learning algorithm [14] is currently the most popular TD-based reinforcement learning algorithm, used also in this work. It learns a Q-function, assigning to each state-action pair (x, a) an estimate of the cumulative discounted sum of future rewards received after executing action a in state x and performing optimally thereafter. The Q-learning update rule, in its generic TD(0) version, may be written as follows:

$$\text{update}^{\beta}\left(Q,\ x_t, a_t,\ r_t + \gamma \max_a Q(x_{t+1}, a) - Q(x_t, a_t)\right), \tag{1}$$

which is the notation used throughout this paper to express that $Q(x_t, a_t)$ is updated using an error value of $r_t + \gamma \max_a Q(x_{t+1}, a) - Q(x_t, a_t)$, to a degree controlled by a step-size parameter β.

The implementation of the update operation given by Equation 1 depends on the function representation method used. For a simple look-up table representation, used in this paper, it simply consists in adding $\beta\Delta$, where Δ is the error value, to the value stored in the appropriate table entry.

Another issue that needs more specification in a practical implementation is how actions to perform at each time step are selected. In the experiments presented in this paper a standard Boltzmann distribution-based stochastic strategy is used, with the selection probability of action a in state x proportional to $\exp(Q(x,a)/T)$, where $T > 0$ is a temperature parameter, adjusting the amount of randomness. This action selection mechanism is known not to perform very well in many cases, but it keeps the algorithms experimented with as generic as possible.

2.3 TTD Procedure

The TTD procedure [2,4] allows one to implement TD-based reinforcement learning algorithms in their TD($\lambda > 0$) versions without conceptually appealing, but computationally demanding eligibility traces [1,10,4]. Only its particular instantiation for the Q-learning algorithm is discussed below, but modifications for other RL algorithms are straightforward [2].

TTD relies on applying, at time t, the following update operation to the state and action from time $t - m + 1$:

$$\text{update}^{\beta}\left(Q,\ x_{t-m+1}, a_{t-m+1},\ z_{t-m+1}^{\lambda,m} - Q(x_{t-m+1}, a_{t-m+1})\right), \tag{2}$$

where $z_t^{\lambda,m}$ is the TTD(λ, m) return for time t, defined as

$$\begin{aligned} z_t^{\lambda,m} = {} & \sum_{k=0}^{m-2} (\gamma\lambda)^k \left[r_{t+k} + \gamma(1-\lambda)\max_a Q(x_{t+k+1}, a)\right] \\ & + (\gamma\lambda)^{m-1}\left[r_{t+m-1} + \gamma \max_a Q(x_{t+m}, a)\right]. \end{aligned} \tag{3}$$

This can be implemented by maintaining an m-step experience buffer, at time t storing records of x_{t-k}, a_{t-k}, r_{t-k}, and $\max_a Q(x_{t-k+1}, a)$ for $k = 0, 1, \ldots, m-1$. For convenience, the corresponding buffer elements are designated by $x_{[k]}$, $a_{[k]}$, $r_{[k]}$, and $u_{[k]}$, respectively. Under this notational convention, the Q-learning version of the TTD(λ, m) procedure is presented in Figure 2, as a sequence of operations performed to process an experience $\langle x, a, r, x' \rangle$. A pair of square brackets appears as an additional argument of the procedure to indicate the indexing mechanism used to access the experience buffer.

```
ttd_procedure$^{\lambda,m}(x, a, r, x', [\,])$:
1. $x_{[0]} := x$; $a_{[0]} := a$; $r_{[0]} := r$; $u_{[0]} := \max_{a'} Q(x', a')$;
2. $z :=$ ttd_return$^{\lambda}([m-1])$;
3. update$^{\beta}\left(Q,\ x_{[m-1]}, a_{[m-1]},\ z - Q(x_{[m-1]}, a_{[m-1]})\right)$;
4. shift the indices of the experience buffer;
5. return $z$.
```

Fig. 2. The TTD procedure.

The operation of Step 2, written as ttd_return$^{\lambda}([m-1])$, computes the TTD(λ, m) return for time $t - m + 1$, i.e., for the least recent experience stored in the buffer (designated by $[m-1]$). This can be performed either iteratively, based directly on the definition of TTD returns, or in an incremental manner, which is particularly efficient. The appropriate algorithms are described in detail in the existing TTD literature [2,4].

3 Combination of TTD and Dyna

The generic Dyna architecture, as presented in Figure 1, does not make any explicit assumptions about the reinforcement learning algorithm used, except that it performs updates based on quadruples consisting of a state, an action, a resulting reward value, and a successor state. In particular, the architecture might be used in the same way for TD-based algorithms, regardless of whether they learn with TD(0) or TD$(\lambda > 0)$. However, while the learning process may benefit from positive λ as usual, it does not have any effect for the planning process, since hypothetical experiences do not form a temporal sequence. This section presents simple modifications of Dyna, based on preliminary ideas first formulated in [5], which use sequences of hypothetical experiences, and thus allow the planning process to benefit from $\lambda > 0$. They rely on applying the TTD procedure to hypothetical experiences similarly as it is applied to real ones.

3.1 Forwards and Backwards Planning with TTD

The most straightforward approach to applying TTD effectively to hypothetical experiences, if they form a temporal sequence, is to process them in exactly the same way as real experiences, using a separate n-element hypothetical experience buffer. Similarly as in the generic Dyna architecture, a forwards model is used to predict for a state-action pair a corresponding reward value and successor state.

Another way of combining Dyna with TTD is to use for planning the TTD returns computed during the regular operation of the procedure (when running for real experiences), rather than maintain a separate experience buffer and perform TTD for hypothetical experiences separately. When, during regular TTD learning at time t, the TTD return for time $t - m + 1$ is computed, it contains meaningful information not only for x_{t-m+1}, but for its possible predecessors, their predecessors, etc. For this idea to be practically useful, a backwards model is needed, that would return for a state x' a triple of $\langle x, a, r\rangle$ such that performing action a in state x is predicted to result in a reinforcement value of r and a successor state x'. The operation of such a model will be referred to as $\mathbb{BM}(x')$.

3.2 TTDyna Algorithm

The two ideas outlined above may be applied together, using two models, a forwards and a backwards one (or a single model providing the functionality of both). The resulting algorithm, called TTDyna, performs one regular TTD update for the current real experiences at each time step, and afterwards it processes several hypothetical experiences in the forwards planning and backwards planning mode. Figure 3 presents the details.

```
At each time step $t$:

1. observe current state $x_t$;
2. $a_t :=$ select_action$(x_t)$;
3. perform action $a_t$; observe new state $x_{t+1}$ and immediate reinforcement $r_t$;
4. update_model$(x_t, a_t, r_t, x_{t+1})$;
5. $z :=$ ttd_procedure$^{\lambda,m}(x_t, a_t, r_t, x_{t+1}, [\,])$;
6. $F := 0$; while $F < N_f$ do
   (a) $F := F + 1$; if $t = 0$ or $f > n_f$ then $f := 0$;
   (b) if $f = 0$ then
         choose a hypothetical state $x$;
   (c) $a :=$ select_action$(x)$;
   (d) if $(r, x') := \mathbb{FM}(x, a)$ then
         i. ttd_procedure$^{\lambda,n}(x, a, r, x', \{\,\})$;
         ii. $x := x'$; $f := f + 1$;
       else $f := 0$.
7. $B := 0$; $b := 0$; while $B < N_b$ do
   (a) $B := B + 1$; if $b > n_b$ then $b := 0$;
   (b) if $b = 0$ then
         $z' := z$; $x' := x_{[m-1]}$;
   (c) if $(x, a, r) := \mathbb{BM}(x')$ then
         i. $u' := \max_{a'} Q(x', a')$; $z' := r + \gamma(\lambda z' + (1 - \lambda)u')$;
         ii. update$^{\beta}(Q,\ x, a,\ z' - Q(x, a))$;
         iii. $x' := x$;
       else $b := 0$.
```

Fig. 3. The TTDyna algorithm.

Steps 5 and 6(d)i refer to the basic TTD algorithm presented in Figure 2, the former for the current real experience, and the latter for a hypothetical experience from the currently processed sequence. As already said above, TTD uses a separate experience buffer for hypothetical experiences, which is denoted using curly braces { } instead of square brackets [ ]. The forwards planning process is controlled by two parameters: N_f is the maximum number of hypothetical experiences to use at a time step, and n_f is the maximum length of a hypothetical experience sequence. If $n_f < N_f$, more than one hypothetical sequence is used at one time step. The sequence started at time t may be continued at subsequent time steps.

The actual number of hypothetical experiences processed during forwards planning and the length of individual sequences may be different from N_f and n_f, respectively, depending on whether the model succeeds at predicting the consequences of state-action pairs it is requested to. The $\mathbb{FM}(x, a)$ operation is assumed to return false whenever the model is unable to give a reliable prediction of a reward and successor state resulting from performing action a in state x. On success, the model is assumed to return a (possibly different on each call) reward and successor state for a given state-action pair. A simple implementation of such a model, used in the experiments presented later in this paper, will be described in Section 3.3.

The backwards planning part of the algorithm performs at each time step a maximum of N_b updates for hypothetical experiences, which may form a sequence with the maximum length $n_b \leq N_b$. Each sequence starts from the (just updated) state $x_{[m-1]}$ and goes backwards in time, along its possible predecessors. Given the TTD return for $x_{[m-1]}$ (assumed to be returned by the call to the TTD procedure in Step 5), their TTD returns may be computed iteratively, as demonstrated by Step 7(c)i. The backwards model is assumed to return false when it is unable to produce a new reliable prediction, and on success to return a (possibly different on each call) state-action-reward triple for the hypothetical successor state for which it is invoked.

3.3 Model Implementation

The algorithm described above strongly depends on model implementation. This in turn is a hard problem itself and deserves separate, extensive studies, which are beyond the scope of this preliminary work. In the experiments presented in this chapter, similarly as for function representation, we use a look-up table model representation, although it may be hypothesized that some generalizing function approximators that have proved themselves useful for the former purpose, may turn out to be useful for the latter as well.

For a forward model, a separate look-up table is used for every action, assuming a discrete action space. Each table entry corresponds to one environment state and stores a predicted reward value r and an estimate p of the occurrence likelihood of each state as a successor state (which also assumes a discrete state space). The operation of model update:

$$\text{update\_model}(x, a, r, x') \tag{4}$$

may be then implemented by

$$\mathrm{FM}_a[x].r := (1-\alpha)\mathrm{FM}_a[x].r + \alpha r \tag{5}$$

and

$$\mathrm{FM}_a[x].p_y := \begin{cases} (1-\alpha)\mathrm{FM}_a[x].p_y + \alpha & \text{if } y = x' \\ (1-\alpha)\mathrm{FM}_a[x].p_y & \text{otherwise,} \end{cases} \tag{6}$$

where α is a step-size parameter, FM_a denotes the look-up table corresponding to action a, brackets are used to indicate for which state the table entry is referred to, and dots to designate access to the two components of each entry. The successor state x', returned by $\mathbb{FM}(x, a)$, may be then chosen based on $\mathrm{FM}_a[x].p$ either deterministically, as $\arg\max_y \mathrm{FM}_a[x].p_y$ (this is the approach adopted in the experiments presented later in this paper), or stochastically, which may be appropriate if there are multiple possible successor states for a given state-action pair.

To predict backwards, a single look-up table may be used, containing, for every state, the corresponding likelihood estimates for the predecessor state and action, and a reward value. The update operations for states and rewards are analogous to those presented above for a forwards model:

$$\mathrm{BM}[x'].p_y := \begin{cases} (1-\alpha)\mathrm{BM}[x'].p_y + \alpha & \text{if } y = x \\ (1-\alpha)\mathrm{BM}[x'].p_y & \text{otherwise,} \end{cases} \tag{7}$$

and

$$\mathrm{BM}[x'].r := (1-\alpha)\mathrm{BM}[x].r + \alpha r. \tag{8}$$

For actions the update operation is a straightforward analog of Equation 6:

$$\mathrm{BM}[x'].q_b := \begin{cases} (1-\alpha)\mathrm{BM}[x'].q_b + \alpha & \text{if } b = a \\ (1-\alpha)\mathrm{BM}[x'].q_b & \text{otherwise,} \end{cases} \tag{9}$$

where q is an estimate of the occurrence likelihood of each action in a preceding experience.

This is indeed a very simple approach to model learning and it has many obvious deficiencies. First, similarly as a tabular function representation, it is only applicable to relatively small tasks. Second, and maybe even more important, in the backwards case it provides no clear way of dealing with states for which there may be multiple predecessor state-action pairs. While a forwards model with such a limitation is still sufficiently powerful for deterministic tasks, for a backwards model it is much more painful. There are some relatively simple possibilities of overcoming these drawbacks, at least in part. One could deal with continuous states by applying the ideas which have already shown themselves useful for function representation in reinforcement learning, i.e., by replacing look-up tables with some kinds of function approximators. In particular, it looks possible to use CMAC-like sparse coarse-coded look-up tables to implement continuous-state models. To allow multiple predecessor state-action pairs for a given state x' in a backwards model, it might be reasonable to first select (e.g., stochastically) a predecessor state x, based on the estimated probability distribution $\mathrm{BM}[x'].p$, and then select an action that, if executed in state x, can be predicted to cause a transition to x'. If there are relatively few actions, which is often the case, one could simply make a forward prediction for x and each possible action, to see whether it can bring the system to state x'. These possible extensions have been, however, postponed for future work, so that this work focuses exclusively on the usefulness of a particular way of using models for reinforcement learning rather than on the usefulness of particular models themselves.

4 TTD Sweeping

Unlike the two TTDyna algorithms, the approach described in this section is intended not to require any models, except for the very simplest one: the memory of a number of past real experiences. This eliminates the problem of choosing an appropriate model representation and learning method and thus may be sometimes more attractive, if it turned out to give similarly good performance.

The basic idea of TTD sweeping, or TTD-SW for short, is to store a relatively large number M of past experiences in the TTD experience buffer, which is made much longer than m, say, several thousand steps instead of one or two dozens. The TTD procedure still operates as usual, on a fraction of the buffer corresponding to the most recent m experiences, but additionally *TTD sweeps* are performed for some other regions of the buffer. A sweep for a buffer region delimited by $k_1 < k_2$ consists in performing sequentially updates for experiences $[k]$, $k = k_1, k_1 + 1, \ldots, k_2$, according to:

$$\mathrm{update}^{\beta}\left(Q,\ x_{[k]}, a_{[k]},\ z_{[k]}^{\lambda,k-k_1+1} - Q(x_{[k]}, a_{[k]})\right), \tag{10}$$

where $z^{\lambda,k-k_1+1}$ denotes the $(k - k_1 + 1)$-step TTD return for experience $[k]$. Figure 4 shows how this process is exactly organized.

Whenever an update of the Q-function for experience $[k]$ is performed, either during regular learning or sweeping, two additional operations take place. First, $u_{[k+1]}$ is updated to reflect the new Q-values for $x_{[k]}$. Second, experience $[k + \mu]$ (which must have been observed μ time steps before experience $[k]$) becomes a candidate for sweeping and its index is inserted into a queue, possibly with some priority value, e.g., depending on the magnitude of the error value used for the update. The maximum queue length, designated by q, limits the number of high-priority experiments which are candidates for further updates.

At each time step a maximum of N_s sweeps are performed, for buffer regions determined as follows. An experience $[s]$ is taken out from the queue. We still assume that the buffer's indices are shifted appropriately at each time step, so

```
At each time step $t$:

1. observe current state $x_t$;
2. $a_t :=$ select_action$(x_t)$;
3. perform action $a_t$; observe new state $x_{t+1}$ and immediate reinforcement $r_t$;
4. ttd_procedure$^{\lambda,m}(x_t, a_t, r_t, x_{t+1}, [\,])$;
5. $u_{[m]} := \max_a Q(x_{[m-1]}, a)$;
6. to_queue$([m - 1 + \mu])$;
7. repeat $N_s$ times
   (a) $[s] :=$ from_queue(); $\hat{\mu} :=$ correct$(\mu)$;
   (b) $z := u_{[s-\hat{\mu}+1]}$;
   (c) for $k = s - \hat{\mu} + 1, s - \hat{\mu} + 2, \ldots, s$ do
       i. $z := r_{[k]} + \gamma(\lambda z + (1 - \lambda)u_{[k]})$;
       ii. update$^{\beta}\left(Q,\ x_{[k]}, a_{[k]},\ z - Q(x_{[k]}, a_{[k]})\right)$;
       iii. $u_{[k+1]} := \max_a Q(x_{[k]}, a)$;
       iv. to_queue$([k + \mu])$.
```

Fig. 4. The TTD sweeping algorithm.

that [0] always refers to the current real experience. Therefore the implementation must provide an appropriate index translation mechanism, to ensure that the index value retrieved from the queue points to the same experience buffer element which was pointed to when the corresponding entry was inserted to the queue. The maximum sweep length is μ, but it may be reduced, which is written $\hat{\mu} = \text{correct}(\mu)$, to ensure that experiences $[s - \hat{\mu}]$ through $[s]$ do belong to the same sequence and that sweeping does not interfere with the regular TTD operation, which involves the first m buffer elements, i.e., $s - \hat{\mu} + 1 \geq m$. Then a sweep is performed for a buffer region delimited by $s - \hat{\mu} + 1$ and s.

5 Experimental Results

In this section the results of preliminary experiments with the algorithms proposed above are presented, designed so as to verify the usefulness of their most generic versions. The performance of the TTDyna and TTD sweeping approaches to planning are compared to each other, as well to the performance of their TD(0) versions and of TD(0) and TTD($\lambda > 0$) without planning.

5.1 Learning Tasks

Three simple grid path-finding tasks, differing in the grid size, are used for the experiments. Figure 5 illustrates the 10×10 grid world which is the basis for these tasks. The state representation supplied to the learning agent is simply the number of the cell it is currently in. At each step the agent can choose any of the four allowed actions of going North, South, East, or West. At the beginning of each trial the agent is placed in its fixed initial location. A trial ends when the agent reaches the goal location. The agent receives a reinforcement value of −1 at all steps except when it enters the goal cell, when the reinforcement is 0.

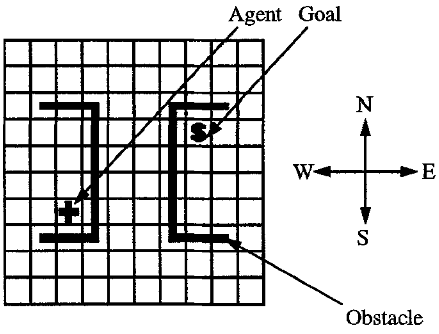


Fig. 5. The grid environment.

Apart from the basic 10×10 task, more difficult 20×20 and 50×50 tasks are used, obtained by dividing each cell (including the barrier and goal cells) of the environment shown in Figure 5 into, respectively, $2^2 = 4$ and $5^2 = 25$ equal smaller cells. The agent's starting location is at $(2,6)$ for the 10×10 task, at $(4,12)$ for the 20×20 task, and at $(12,32)$ for the 50×50 task, assuming that the coordinates count from 0 and start from the left upper corner. The shortest path to any goal cell for the three tasks consists, respectively, of 20 steps, 36 steps, and 84 steps.

5.2 Experimental Design and Results

The following algorithms were applied to each of the three grid tasks:

- the TTD procedure with $\lambda = 0$ and $\lambda = 0.5$,
- TTDyna with $\lambda = 0$ and $\lambda = 0.5$,
- TTD sweeping with $\lambda = 0$ and $\lambda = 0.5$.

The TTD procedure used $m = 25$. For all algorithms γ was set to 0.95 and T to 0.01. These values appeared reasonable based on prior work with TTD [2,4]. Additionally, for TTDyna we used $N_f = 10$, $n_f = 5$, and $N_b = n_b = 5$, and for TTD sweeping $M = 500$, $N_s = 3$, $\mu = 8$, and a priority queue of length $q = 25$. A number of preliminary runs with different settings of these planning parameters were carried out and the above were found to be roughly best, but not crucial for successful performance. Both the algorithms performed similarly for a wide range of parameter values. The step-size parameters used for updating the Q-function were equal 0.5 for $\lambda = 0$ and 0.125 for $\lambda = 0.5$, the values optimized in a series of preliminary runs. Twice larger or smaller β appeared to yield only slightly worse results. Model update operations for TTDyna, implemented according to

Equations 5 and 6, used $\alpha = 1$, which is a natural choice for deterministic tasks. The Q-values were initialized to -5 for the 10×10 task, -7 for the 20×20 task, and -10 for the 50×50 task, to provide a reliable initial guess.

Figure 6 compares the learning curves obtained by using TTDyna, TTD-SW, and "pure" TTD for the two λ values tried. The results are averaged over 25 independent experimental runs. Table 1 presents the total number of real interactions performed by each of the algorithms before convergence. To better illustrate how the learning speed of the investigated algorithms scales up with the size of the state space, Table 2 presents the factors by which the numbers of steps and trials necessary to converge for all the algorithms in the 20×20 and 50×50 tasks are greater than in the 10×10 task.

Table 1. The average numbers of real interactions before convergence.

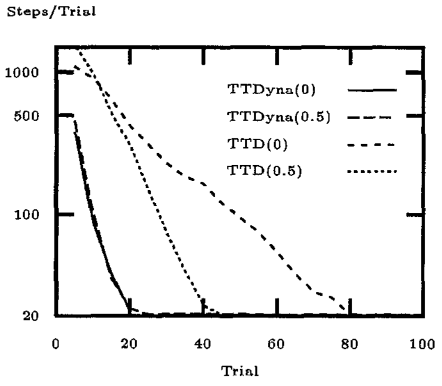
Task/Algorithm	10 × 10	20 × 20	50 × 50 task
TTD(0)	22,381	132,971	1,938,440
TTD(0.5)	18,626	114,651	1,821,950
TTDyna(0)	2753	18,964	264,970
TTDyna(0.5)	3303	20,063	247,705
TTD-SW(0)	7404	52,470	716,453
TTD-SW(0.5)	9885	59,039	667,253

Table 2. The factors by which the numbers of steps and trials before convergence for the 20×20 and 50×50 tasks are greater than for the 10×10 task.

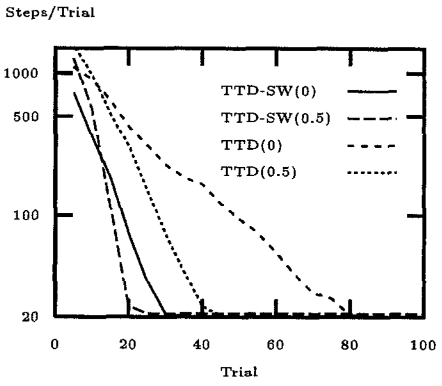
Task/Algorithm	20 × 20		50 × 50	
	Steps	Trials	Steps	Trials
TTD(0)	5.95	3.15	86.5	22.0
TTD(0.5)	6.2	2.45	98.0	16.0
TTDyna(0)	6.9	2.8	97.5	18.0
TTDyna(0.5)	6.05	1.65	75.5	8.3
TTD-SW(0)	7.05	2.75	98.0	16.5
TTD-SW(0.5)	5.95	2.2	67.0	12.0

We can observe that

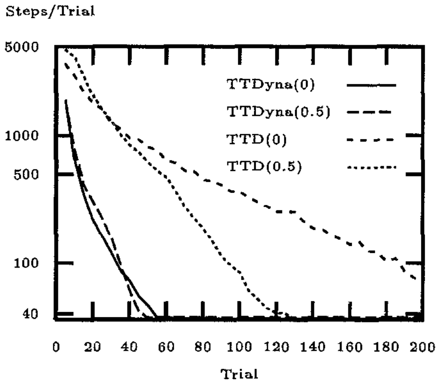
- without planning, $\lambda > 0$ considerably improves convergence speed,
- for $\lambda = 0$, both TTDyna and TTD-SW result in much faster learning than for "pure" TD(0),
- $\lambda > 0$ gives a significant further learning speed improvement for TTD-SW,
- unlike in the 10×10 task, for the larger tasks the learning speed of TTDyna improves for positive λ,



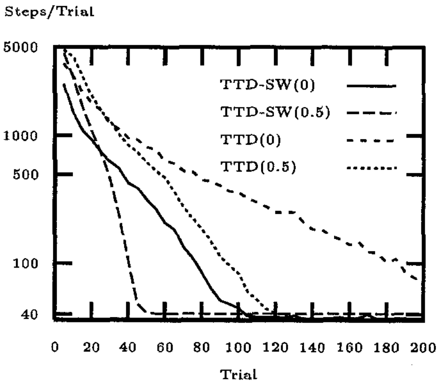
(a) TTDyna, 10 × 10 task



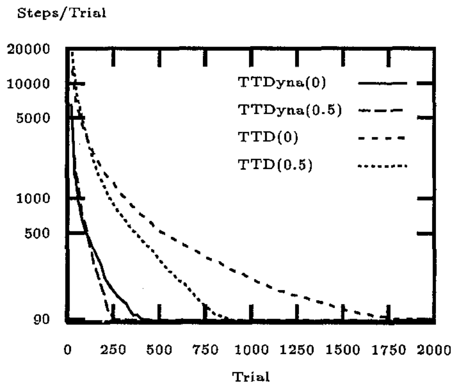
(b) TTD-SW, 10 × 10 task



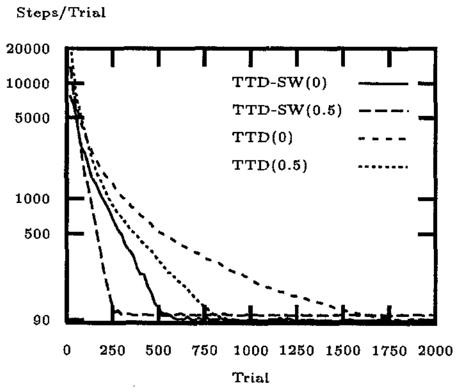
(c) TTDyna, 20 × 20 task



(d) TTD-SW, 20 × 20 task



(e) TTDyna, 50 × 50 task



(f) TTD-SW, 50 × 50 task

Fig. 6. Learning curves for TTD, TTDyna, and TTD-SW.

- in general, the improvements due to planning and $\lambda > 0$ are clearly more significant for the two larger tasks than for the 10×10 one,
- the advantages of TTDyna and TTD-SW over TTD are particularly noticeable when one compares the number of (real) time steps necessary to converge,
- while $\lambda > 0$ reduces the number of trials until convergence for both TTD and the two planning algorithms, it does not always reduce (and may even increase) the number of steps (except for TTD), because the first few trials are, on the average, longer than for $\lambda = 0$,
- with respect to the number of trials necessary to converge, TTDyna and TTD-SW scale up noticeably better with the size of the state space than non-planning TTD, and using positive λ further improves their scaling properties,
- the effects of planning and positive λ on scaling with respect to the number of steps necessary to converge is not quite clear, but the combination of TTDyna and TTD-SW with $\lambda > 0$ appears to give some improvement over TTDyna(0) and TTD-SW(0), as well as over TTD($\lambda > 0$).

6 Related Work

The two presented algorithms borrow heavily from prior work on model-based reinforcement learning and on TD(λ). The TTDyna algorithm is a combination of Sutton's [12] Dyna architecture with TTD [2]. Its novelty consists essentially in a special way of using hypothetical experiences, so that they form temporal sequences and thus can be reasonably processed by TTD. The TTD sweeping algorithm is related to the ***prioritized sweeping*** technique of Moore and Atkeson [8] and a similar *Queue-Dyna* algorithm of Peng and Williams [9] on one hand, and Lin's [6] *experience replay*.

The primary improvement of prioritized sweeping (and Queue-Dyna) over the original Dyna algorithm is that hypothetical experiences to process are chosen in a special way, according to their heuristically estimated usefulness. The model used is no longer a simple forwards model: it additionally stores each state's predecessors, as in a backwards model. States have also certain priority values assigned and at each time step K states with the highest priorities are used to generate hypothetical experiences by applying the model. Whenever a state's utility is updated with a large error value, the priorities of its possible predecessors are increased appropriately (to a degree proportional to the corresponding estimated transition probabilities). This idea is adopted by TTD sweeping, although used in a very simplified form: no model is maintained, except for a memory of a number of past time steps. It would be interesting and possibly useful to extend the TTD-SW algorithm, so that it handles multiple potential predecessors of each state as well.

In experience replay, as used by Lin [6], after completing a trial a number of past experiences are processed in the reversed chronological order. This is actually the same as a single sweep of the TTD-SW algorithm. Thus, TTD sweeping is in effect a combination of experience replay with a Dyna-like control strategy, which determines which experiences it is most useful to replay.

7 Conclusion

This paper investigated some possible ways of implementing the well-known idea of augmenting reinforcement learning by model-based planning in a more effective way. The motivation was to make the planning process benefit from TD($\lambda > 0$) in a similar way as learning. The TTD procedure, which implements TD(λ) without eligibility traces, was found particularly useful for this purpose.

The presented experimental results show that the two proposed techniques indeed reduce the necessary number of real experiences necessary of converge by a large factor, at least several times. TTDyna appeared to perform better than TTD-SW, though an attractive feature of the latter is that it does not need any explicit models. However, given more carefully designed and reliable models than the simple ones used in the experiments, the advantages of TTDyna are likely to be even more evident.

The learning speed improvement due to TTDyna and TTD-SW is particularly noticeable in comparison to TD(0) without planning. It is apparently less spectacular in comparison to TTD($\lambda > 0$) without planning, because $\lambda > 0$ alone gives much faster learning. However, when one compares the number of (real) time steps before convergence rather than the number of trials, it turns out that the effects of planning are definitely more significant than the effects of positive λ. For the planning using $\lambda > 0$ appeared not to give any further reduction of the number of steps necessary to converge, because the first few trials, before any meaningful Q-values have been learned, were relatively longer than for $\lambda = 0$, which is certainly somewhat disappointing. This effect may be due to the rather *ad hoc* settings of other parameters and the primitive action selection mechanism, and thus there is a chance that it will be eventually overcome. Positive λ was found to be still useful for the planning process anyway, as it may give some further reduction of the necessary number of trials, particularly for large tasks. In many cases minimizing the number of trials may be equally or more important as minimizing the number of steps. It is therefore important to note that, with respect to the number of trials necessary to converge, the proposed algorithms appear to scale better with the size of the state space than TTD without planning, particularly for $\lambda > 0$.

It must be stressed that the reported results have been obtained by using the most generic versions of the algorithms. It seems likely that much greater performance gain can be obtained by using better model representation methods for TTDyna and better TTD sweeping heuristics. This is what should be verified by future work.

Acknowledgements

The author gratefully acknowledges the financial support from the Polish Committee for Scientific Research under Grant 8 T11C 04611 and from the Foundation for Polish Science.

References

1. A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:835–846, 1983.
2. P. Cichosz. Truncating temporal differences: On the efficient implementation of TD(λ) for reinforcement learning. *Journal of Artificial Intelligence Research*, 2:287–318, 1995.
3. P. Cichosz. Truncated temporal differences with function approximation: Successful examples using CMAC. In *Proceedings of the Thirteenth European Symposium on Cybernetics and Systems Research (EMCSR-96)*, 1996.
4. P. Cichosz and J. J. Mulawka. Fast and efficient reinforcement learning with truncated temporal differences. In *Proceedings of the Twelfth International Conference on Machine Learning (ML-95)*. Morgan Kaufmann, 1995.
5. P. Cichosz and J. J. Mulawka. Integrated architectures for learning, planning, and reacting based on approximating TD(λ). In *Proceedings of the First International Workshop on Intelligent Adaptive Systems (IAS-95)*, 1995.
6. Long-Ji Lin. *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, School of Computer Science, Carnegie-Mellon University, January 1993.
7. S. Mahadevan and J. Connell. Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence*, 55:311–365, 1992.
8. A. W. Moore and C. G. Atkeson. Prioritized sweeping: Reinforcement learning with less memory and less time. *Machine Learning*, 13:103–130, 1993.
9. J. Peng and R. J. Williams. Efficient learning and planning within the Dyna framework. In *Proceedings of the Second International Conference on Simulation of Adaptive Behavior*. The MIT Press, 1993.
10. R. S. Sutton. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, Department of Computer and Information Science, University of Massachusetts, 1984.
11. R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.
12. R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning (ML-90)*. Morgan Kaufmann, 1990.
13. R. S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*. Morgan Kaufmann, 1996.
14. C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, 1989.