Sample-Based Learning and Search with Permanent and Transient Memories

David Silver SILVER@CS.UALBERTA.CA
Richard S. Sutton SUTTON@CS.UALBERTA.CA
Martin Müller MMUELLER@CS.UALBERTA.CA
Department of Computing Science, University of Alberta, Edmonton, Alberta

Abstract

We present a reinforcement learning architecture, Dyna-2, that encompasses both sample-based learning and sample-based search, and that generalises across states during both learning and search. We apply Dyna-2 to high performance Computer Go. In this domain the most successful planning methods are based on sample-based search algorithms, such as UCT, in which states are treated individually, and the most successful learning methods are based on temporal-difference learning algorithms, such as Sarsa, in which linear function approximation is used. In both cases, an estimate of the value function is formed, but in the first case it is transient, computed and then discarded after each move, whereas in the second case it is more permanent, slowly accumulating over many moves and games. The idea of Dyna-2 is for the transient planning memory and the permanent learning memory to remain separate, but for both to be based on linear function approximation and both to be updated by Sarsa. To apply Dyna-2 to 9×9 Computer Go, we use a million binary features in the function approximator, based on templates matching small fragments of the board. Using only the transient memory, Dyna-2 performed at least as well as UCT. Using both memories combined, it significantly outperformed UCT. Our program based on Dyna-2 achieved a higher rating on the Computer Go Online Server than any handcrafted or traditional search based program.

Appearing in *Proceedings of the 25th International Conference on Machine Learning*, Helsinki, Finland, 2008.

1. Introduction

Reinforcement learning can be subdivided into two fundamental problems: *learning* and *planning*. Informally, the goal of learning is for an agent to improve its policy from its interactions with the environment. The goal of planning is for an agent to improve its policy without further interaction with its environment. The agent can deliberate, reason, ponder, think or search, so as to find the best behaviour in the available computation time. Sample-based methods can be applied to both problems. During learning, the agent samples experience from the real world: it executes an action at each time-step and observes its consequences. During planning, the agent samples experience from a *model* of the world: it *simulates* an action at each computational step and observes its consequences. We propose that an agent can both learn and plan effectively using sample-based reinforcement learning algorithms. We use the game of 9×9 Go as an example of a large-scale, high-performance application in which learning and planning both play significant roles.

In the domain of Computer Go, the most successful learning methods have used sample-based reinforcement learning to extract domain knowledge from games of self-play (Schraudolph et al., 1994; Dahl, 1999; Enzenberger, 2003; Silver et al., 2007). The value of a position is approximated by a multi-layer perceptron, or a linear combination of binary features, that form a compact representation of the state space. Temporal difference learning is used to update the value function, slowly accumulating knowledge from the complete history of experience.

The most successful planning methods use sample-based search to identify the best move in the current position. 9×9 Go programs based on the UCT algorithm (Kocsis & Szepesvari, 2006) have now achieved master level (Gelly & Silver, 2007; Coulom, 2007). The UCT algorithm begins each new move with no domain

knowledge, but rapidly learns the values of positions in a temporary search tree. Each state in the tree is explicitly represented, and the value of each state is learned by Monte-Carlo simulation, from games of self-play that start from the current position.

In this paper we develop an architecture, Dyna-2, that combines these two approaches. Like the Dyna architecture (Sutton, 1990), the agent updates a value function both from real experience, and from simulated experience that is sampled using a model of the world. The new idea is to maintain two separate memories: a permanent learning memory that is updated from real experience; and a transient planning memory that is updated from simulated experience. Both memories use linear function approximation to form a compact representation of the state space, and both memories are updated by temporal-difference learning.

2. Reinforcement Learning

We consider sequential decision-making processes, in which at each time-step t the agent receives a state s_t, executes an action a_t according to its current policy $\pi_t(s, a)$, and then receives a scalar reward r_{t+1}.

2.1. Sample-Based Learning

Most efficient reinforcement learning methods use a *value function* as an intermediate step for computing a policy. In episodic tasks the action-value function $Q^\pi(s, a)$ is the expected total reward from state s after taking action a and then following policy π.

In large domains, it is not possible or practical to learn a value for each individual state. In this case, it is necessary to approximate the value function using features $\phi(s, a)$ and parameters θ. A simple and successful approach (Sutton, 1996) is to use a linear function approximation $Q(s, a) = \phi(s, a)^T\theta$. We note that table lookup is a special case of linear function approximation, using binary features $\phi(s, a) = e(s, a)$, where $e(s, a)$ is a unit vector with a one in the single component corresponding to (s, a) and zeros elsewhere.

The TD(λ) algorithm (Sutton, 1988) estimates the value of the current state from the value of subsequent states. The λ parameter determines the temporal span over which values are updated. At one extreme, TD(0) bootstraps the value of a state from its immediate successor. At the other extreme, TD(1) updates the value of a state from the final return; it is equivalent to Monte-Carlo evaluation (Sutton & Barto, 1998). TD(λ) can be incrementally computed by maintaining a vector of *eligibility traces* z_t.

The Sarsa algorithm (Rummery & Niranjan, 1994) combines temporal difference evaluation with policy improvement. An action-value function is estimated by the TD(λ) algorithm, and the policy is improved by selecting actions according to an ϵ-greedy policy. The action-value function is updated from each tuple (s, a, r, s', a') of experience, using the TD(λ) update rule,

$$\delta_t = r_{t+1} + Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \quad (1)$$

$$z_t = \lambda z_{t-1} + \phi(s_t, a_t) \quad (2)$$

$$\theta_t = \theta_{t-1} + \alpha\delta_t z_t(s) \quad (3)$$

2.2. Sample-Based Search

Sample-based planning applies sample-based reinforcement learning methods to simulated experience. This requires a sample model of the world: a state transition generator $A_t(s, a) \in \mathcal{S} \times \mathcal{A} \mapsto \mathcal{S}$ and reward generator $B_t(s, a) \in \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$. The effectiveness of sample-based planning depends on the accuracy of the model (Paduraru, 2007). In *sample-based search*, experience is simulated from the real state s, so as to identify the best action from this state.

Monte-Carlo simulation is a simple but effective method for sample-based search. Multiple episodes are simulated, starting from the real state s, and following a random policy. The action-values $Q(s, a)$ are estimated by the empirical average of the returns of all episodes in which action a was taken from the real state s. After simulation is complete, the agent selects the greedy action $\text{argmax}_a\, Q(s, a)$, and proceeds to the next real state.

Monte-Carlo tree search constructs a search tree containing all state–action pairs that have been visited by the agent. Each simulation consists of two distinct phases: greedy action selection while within the tree, and then random action selection until termination. If a simulated state s is fully represented in the search tree, i.e. all actions from s have already been tried, then the agent selects the greedy action $\text{argmax}_a\, Q(s, a)$. Otherwise, the agent selects actions at random. After each simulation, the action-values $Q(s, a)$ of all states and actions experienced in the episode are updated to the empirical average return following each state–action pair. In practice, only one new state–action pair is added per episode, resulting in a tree-like expansion.

The UCT algorithm (Kocsis & Szepesvari, 2006) improves the greedy action selection in Monte-Carlo tree search. Each state of the search tree is treated as a multi-armed bandit, and actions are chosen using the UCB algorithm for balancing exploration and exploita-

tion (Auer et al., 2002).

2.3. Dyna

The Dyna architecture (Sutton, 1990) combines sample-based learning with sample-based planning. The agent learns a model of the world from real experience, and updates its action-value function from both real and simulated experience. Before each real action is selected, the agent performs some sample-based planning. For example, the Dyna-Q algorithm remembers all previous states, actions and transitions. During planning, experience is simulated by sampling states, actions and transitions from the empirical distribution. A Q-learning update is applied to update the action-value function after each sampled transition, and after each real transition.

2.4. Tracking

Traditional learning methods focus on finding a single best solution to the learning problem. In reinforcement learning one may seek an algorithm that converges on the optimal value function (or optimal policy). However, in large domains the agent may not have sufficient resources to perfectly represent the optimal value function. In this case we can actually achieve better performance by *tracking* the current situation rather than *converging* on the best overall parameters. The agent can specialise its value function to its current region of the state space, and update its representation as it moves through the state space. The potential for specialisation means that tracking methods may outperform converging methods, even in stationary domains (Sutton et al., 2007).

3. Permanent and Transient Memories

We define a *memory* to be the set of features and corresponding parameters used by an agent to estimate the value function. In our architecture, the agent maintains two distinct memories: a *permanent memory* (ϕ, θ) updated during sample-based learning, and a *transient memory* $(\bar{\phi}, \bar{\theta})$ updated during sample-based search. The value function is a linear combination of the transient and permanent memories, such that the transient memory tracks a local correction to the permanent memory,

$$Q(s,a) = \phi(s,a)^T\theta \quad (4)$$

$$\bar{Q}(s,a) = \phi(s,a)^T\theta + \bar{\phi}(s,a)^T\bar{\theta} \quad (5)$$

where $Q(s,a)$ is a permanent value function, and $\bar{Q}(s,a)$ is a combined value function.

We refer to the distribution of states and actions encountered during real experience as the *learning distribution*, and the distribution encountered during simulated experience as the *search distribution*. The permanent memory is updated from the learning distribution and converges on the best overall representation of the value function, based on the agent's past experience. The transient memory is updated from the search distribution and tracks the local nuances of the value function, based on the agent's expected future experience.

4. Dyna-2

Algorithm 1 Episodic Dyna-2

procedure LEARN
  Initialise A, B ▷ Transition and reward models
  $\theta \leftarrow 0$ ▷ Clear permanent memory
  loop
    $s \leftarrow s_0$ ▷ Start new episode
    $\bar{\theta} \leftarrow 0$ ▷ Clear transient memory
    $z \leftarrow 0$ ▷ Clear eligibility trace
    SEARCH(s)
    $a \leftarrow \pi(s; \bar{Q})$ ▷ e.g. ϵ-greedy
    while s is not terminal **do**
      Execute a, observe reward r, state s'
      $(A, B) \leftarrow$ UPDATEMODEL(s, a, r, s')
      SEARCH(s')
      $a' \leftarrow \pi(s'; \bar{Q})$
      $\delta \leftarrow r + Q(s', a') - Q(s, a)$ ▷ TD-error
      $\theta \leftarrow \theta + \alpha(s, a)\delta z$ ▷ Update weights
      $z \leftarrow \lambda z + \phi$ ▷ Update eligibility trace
      $s \leftarrow s', a \leftarrow a'$
    end while
  end loop
end procedure

procedure SEARCH(s)
  while time available **do**
    $\bar{z} \leftarrow 0$ ▷ Clear eligibility trace
    $a \leftarrow \bar{\pi}(s; \bar{Q})$ ▷ e.g. ϵ-greedy
    while s is not terminal **do**
      $s' \leftarrow A(s, a)$ ▷ Sample transition
      $r \leftarrow B(s, a)$ ▷ Sample reward
      $a' \leftarrow \bar{\pi}(s'; \bar{Q})$
      $\bar{\delta} \leftarrow r + \bar{Q}(s', a') - \bar{Q}(s, a)$ ▷ TD-error
      $\bar{\theta} \leftarrow \bar{\theta} + \bar{\alpha}(s, a)\bar{\delta}\bar{z}$ ▷ Update weights
      $\bar{z} \leftarrow \bar{\lambda}\bar{z} + \bar{\phi}$ ▷ Update eligibility trace
      $s \leftarrow s', a \leftarrow a'$
    end while
  end while
end procedure

The Dyna-2 architecture can be summarised as Dyna with Sarsa updates, permanent and transient memories, and linear function approximation (see Algorithm 1). The agent updates its permanent memory from real experience. Before selecting a real action, the agent executes a sample-based search from the current state. The search procedure simulates complete episodes from the current state, sampled from the model, until no more computation time is available. The transient memory is updated during these simulations to learn a local correction to the permanent memory; it is cleared at the beginning of each real episode.

A particular instance of Dyna-2 must specify learning parameters: a policy π to select real actions; a set of features ϕ for the permanent memory; a temporal difference parameter λ; and a learning rate $\alpha(s, a)$. Similarly, it must specify the equivalent search parameters: a policy $\bar{\pi}$ to select actions during simulation; a set of features $\bar{\phi}$ for the transient memory; a temporal difference parameter $\bar{\lambda}$; and a learning rate $\bar{\alpha}(s, a)$.

The Dyna-2 architecture subsumes a large family of learning and search algorithms. If there is no transient memory, $\bar{\phi} = \emptyset$, then the search procedure has no effect and may be skipped. This results in the linear Sarsa algorithm.

If there is no permanent memory, $\phi = \emptyset$, then Dyna-2 reduces to a sample-based search algorithm. For example, Monte-Carlo tree search is achieved by choosing table lookup $\bar{\phi}(s, a) = e(s, a)$[1]; using a simulation policy that is greedy within the tree, and then uniform random until termination; and selecting learning parameters $\bar{\lambda} = 1$ and $\bar{\alpha}(s, a) = 1/n(s, a)$, where $n(s, a)$ counts the number of times that action a has been selected in state s. The UCT algorithm replaces the greedy phase of the simulation policy with the UCB rule for action selection.

Finally, we note that real experience may be accumulated offline prior to execution. Dyna-2 may be executed on any suitable training environment (e.g. a helicopter simulator) before it is applied to real data (e.g. a real helicopter). The permanent memory is updated offline, but the transient memory is updated online. Dyna-2 provides a principled mechanism for combining offline and online knowledge(Gelly & Silver, 2007); the permanent memory provides prior knowledge and a baseline for fast learning. Our examples of Dyna-2 in Computer Go operate in this manner.

[1]The number of entries in the table can increase over time, to give a tree-like expansion.

5. Dyna-2 in Computer Go

In domains with spatial coherence, binary features can be constructed to exploit spatial structure at multiple levels (Sutton, 1996). The game of Go exhibits strong spatial coherence: expert players describe positions using a broad vocabulary of shapes (Figure 1a). A simple way to encode basic shape knowledge is through a large set of *local shape features* which match a particular configuration within a small region of the board (Silver et al., 2007). We define the feature vector $\phi^{square(m)}$ to be the vector of local shape features for $m \times m$ square regions, for all possible configurations and square locations. For example, Figure 1a shows several local shape features of size 3×3. Combining local shape features of different sizes builds a representation spanning many levels of generality: we define the multi-level feature vector $\phi^{square(m,n)} = [\phi^{square(m)}; ...; \phi^{square(n)}]$. In 9×9 Go there are nearly a million $\phi^{square(1,3)}$ features, about 200 of which are non-zero at any given time.

Local shape features can be used as a permanent memory, to represent general domain knowledge. For example, local shape features can be learned offline, using temporal difference learning and training by self-play (Silver et al., 2007; Gelly & Silver, 2007). However, local shape features can also be used as a transient memory[2], by learning online from simulations from the current state. The representational power of local shape features is significantly increased when they can track the short-term circumstances (Sutton et al., 2007). A local shape may be bad in general, but good in the current situation (Figure 1b). By training from simulated experience, starting from the current state, we can focus learning on what works well *now*.

We apply the Dyna-2 algorithm to 9×9 Computer Go using local shape features $\phi(s, a) = \bar{\phi}(s, a) = \phi^{square(1,3)}(s \circ a)$, where $s \circ a$ indicates the *afterstate* following action a in state s (Sutton & Barto, 1998). We use a self-play model, an ϵ-greedy policy, and default parameters of $\lambda = \bar{\lambda} = 0.4$, $\alpha(s, a) = 0.1/|\phi(s, a)|$, $\bar{\alpha}(s, a) = 0.1/|\bar{\phi}(s, a)|$, and $\epsilon = 0.3$. We modify the Dyna-2 algorithm slightly to utilise the logistic function and to minimise a cross-entropy loss function, by replacing the value function approximation in (4) and (5),

$$Q(s, a) = \sigma(\phi(s, a)^T \theta) \quad (6)$$

$$\bar{Q}(s, a) = \sigma(\phi(s, a)^T \theta + \bar{\phi}(s \circ a)^T \bar{\theta}) \quad (7)$$

where $\sigma(x) = \frac{1}{1+e^{-x}}$ is the logistic function.

[2]Symmetric local shape features share weights in the permanent memory, but not in the transient memory.

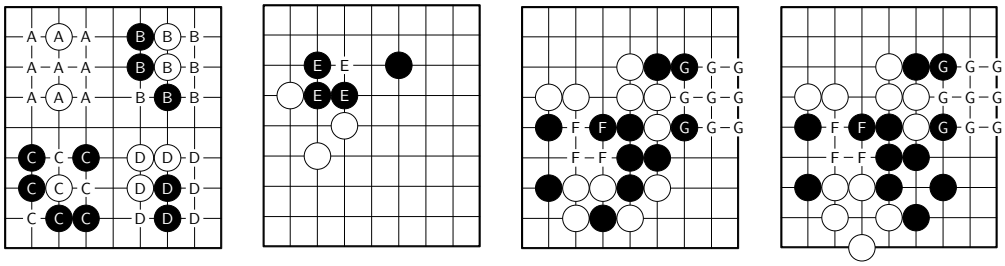

Figure 1. a) Examples of 3×3 local shape features, matching common shapes known to Go players: the *one-point jump* (A), *hane* (B), *net* (C) and *turn* (D). b) The *empty triangle* (E) is normally considered a bad shape; it can be learned by the permanent memory. However, in this position the empty triangle makes a good shape, known as *guzumi*; it can be learned by the transient memory. c) White threatens to cut blacks stones apart at F and G. 2×2 and 3×3 local shape features can represent the local consequences of cutting at F and G respectively. d) A position encountered when searching from (c): Dyna-2 is able to generalise, using local shape features in its transient memory, and can re-use its knowledge about cutting at F and G. UCT considers each state uniquely, and must re-search each continuation.

In addition we ignore local shape features consisting of entirely empty intersections; we clear the eligibility trace for exploratory actions; and we use the default policy described in (Gelly et al., 2006) after the first $D = 10$ moves of each simulation. We refer to the complete algorithm as *Dyna-2-Shape*, and implement this algorithm in our program *RLGO*, which executes almost 2000 complete episodes of simulation per second on a 3 GHz processor.

For comparison, we implemented the UCT algorithm, based on the description in (Gelly et al., 2006). We use an identical default policy to the Dyna-2-Shape algorithm, to select moves when outside of the search tree, and a first play urgency of 1. We evaluate both programs by running matches against GnuGo, a standard benchmark program for Computer Go.

We compare the performance of local shape features in the permanent memory alone; in the transient memory alone; and in both the permanent and transient memories. We also compare the performance of local shape features of different sizes (see Figure 3). Using only the transient memory, Dyna-2-Shape outperformed UCT by a small margin. Using Dyna-2-Shape with both permanent and transient memories provided the best results, and outperformed UCT by a significant margin.

Local shape features would normally be considered naive in the domain of Go: the majority of shapes and tactics described in Go textbooks span considerably larger regions of the board than 3×3 squares. Indeed, when used only in the permanent memory, the local shape features win just 5% of games against GnuGo. However, when used in the transient memory, even the $\phi^{square(1,2)}$ features achieve performance comparable to UCT. Unlike UCT, the transient memory can generalise in terms of local responses: for example, it quickly learns the importantance of black connecting when white threatens to cut (Figures 1c and 1d).

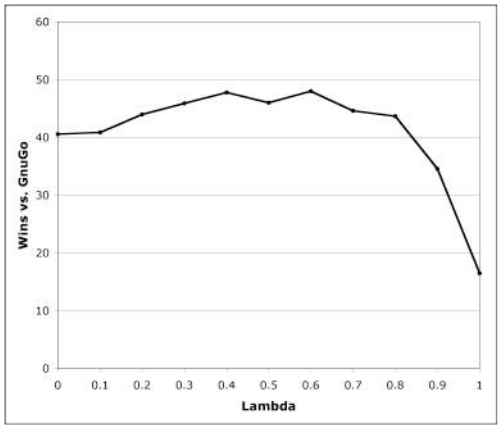

Figure 2. Winning rate of RLGO against GnuGo 3.7.10 (level 0) in 9×9 Go, using Dyna-2-Shape with 1000 simulations/move, for different values of $\bar{\lambda}$. Each point represents the winning percentage over 1000 games.

We also study the effect of the temporal difference parameter $\bar{\lambda}$ in the search procedure (see Figure 2). We see that bootstrapping ($\bar{\lambda} < 1$) provides significant benefits. Previous work in sample-based search has largely been restricted to Monte-Carlo methods (Tesauro & Galperin, 1996; Kocsis & Szepesvari, 2006; Gelly et al., 2006; Gelly & Silver, 2007; Coulom, 2007). Our results suggest that generalising these approaches to temporal difference learning methods may provide significant benefits when value function approximation is used.

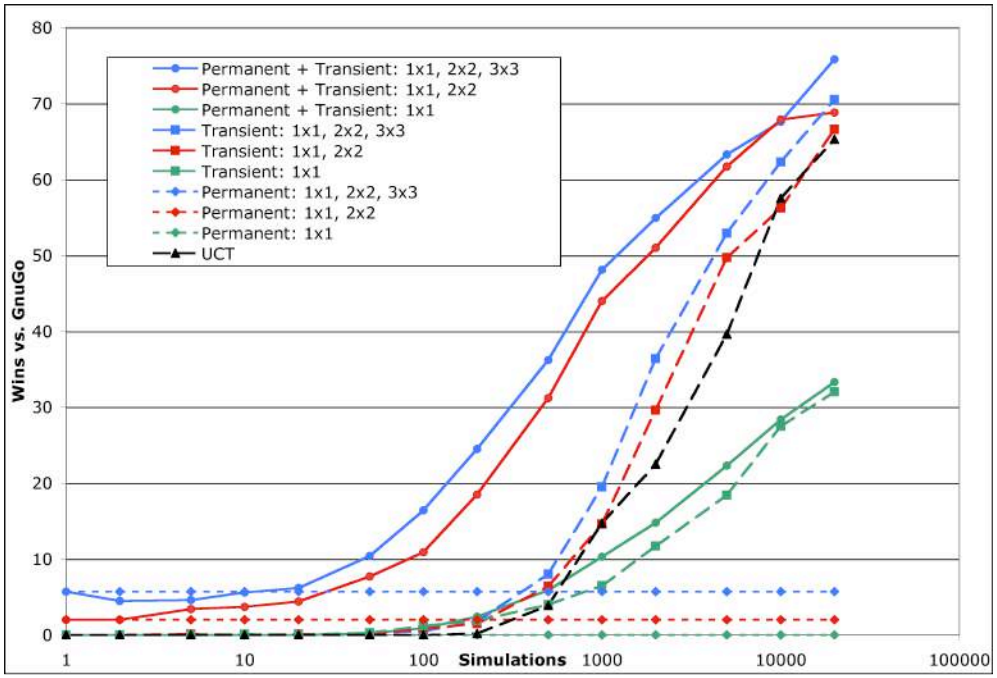

Figure 3. Winning rate of RLGO against GnuGo 3.7.10 (level 0) in 9×9 Go, using Dyna-2-Shape for different simulations/move. Local shape features are used in either the permanent memory (dotted lines), the transient memory (dashed lines), or both memories (solid lines). The permanent memory is trained offline from 100,000 games of self-play. Local shape features varied in size from 1×1 up to 3×3. Each point represents the winning percentage over 1000 games.

6. Dyna-2 and Heuristic Search

In games such as Chess, Checkers and Othello, master level play has been achieved by combining a heuristic evaluation function with α-β search. The heuristic is typically approximated by a linear combination of binary features, and can be learned offline by temporal-difference learning and self-play (Baxter et al., 1998; Schaeffer et al., 2001; Buro, 1999). Similarly, in the permanent memory of our architecture, the value function is approximated by a linear combination of binary features, learned offline by temporal-difference learning and self-play (Silver et al., 2007). Thus it is natural to compare Dyna-2 with approaches based on α-β search.

Dyna-2 combines a permanent memory with a transient memory, using sample-based search. In contrast, the classical approach uses the permanent memory $Q(s, a)$ as an evaluation function for α-β search. A hybrid approach is also possible, in which the combined value function $\bar{Q}(s, a)$ is used as an evaluation function for α-β search, including both permanent and transient memories. This can be viewed as searching with a dynamic evaluation function that evolves according to the current context. We compare all three approaches in Figure 4.

Dyna-2 outperformed classical search by a wide margin. In the game of Go, the consequences of a particular move (for example, playing good shape as in Figure 1a) may not become apparent for tens or even hundreds of moves. In a full-width search these consequences remain beyond the horizon, and will only be recognised if represented by the evaluation function. In contrast, sample-based search only uses the permanent memory as an initial guide, and learns to identify the consequences of particular patterns in the current situation. The hybrid approach successfully combines this knowledge with the precise lookahead provided by full-width search.

Using the hybrid approach, our program RLGO established an Elo rating of 2130 on the Computer Go Online Server, more than any handcrafted or traditional search program.

7. Related work

The Computer Go program MoGo uses the heuristic UCT algorithm (Gelly & Silver, 2007) to achieve *dan*-level performance. This algorithm can be viewed as an instance of Dyna-2 with local shape features in the permanent memory, and table lookup in the transient memory. It uses a step-size of $\bar{\alpha}(s, a) =$

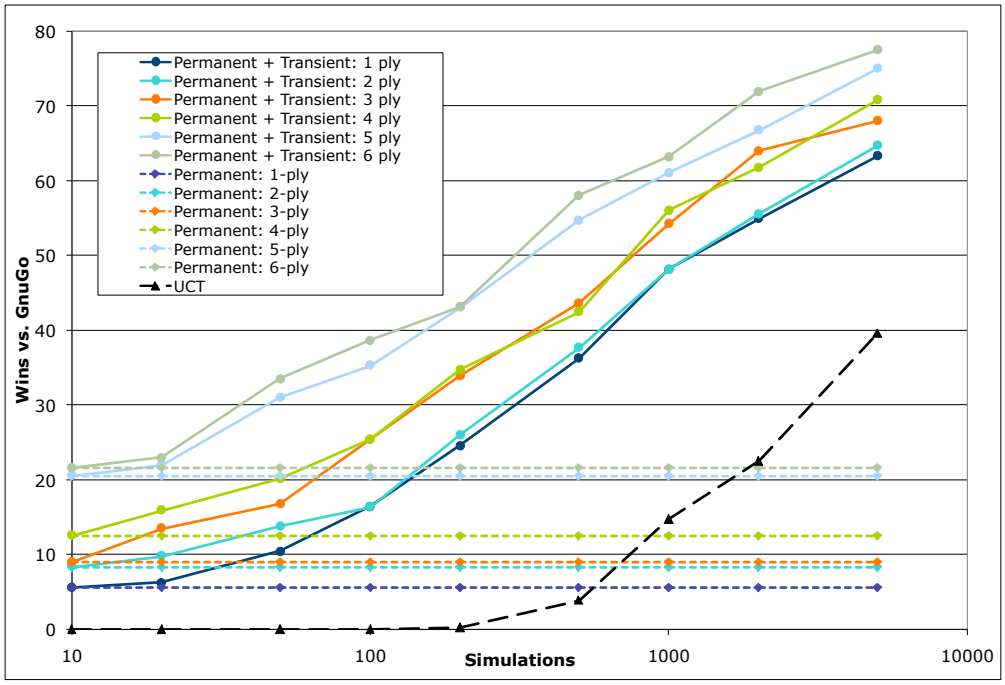

Figure 4. Winning rate of RLGO against GnuGo 3.7.10 (level 0) in 9×9 Go, using Dyna-2-Shape. A full-width α-β search is used for move selection, using a value function based on either the permanent memory (dotted lines), or both memories (solid lines). A 1-ply search corresponds to the usual move selection procedure in Dyna-2. For comparison, a 5-ply search takes approximately the same computation time as 1000 simulations. The permanent memory is trained offline from 100,000 games of self-play. Each point represents the winning percentage over 1000 games.

$1/(n_{prior}(s,a)+n(s,a))$. The confidence in the permanent memory is specified by n_{prior} in terms of *equivalent experience*, i.e. the worth of the permanent memory, measured in episodes of simulated experience.

In addition, MoGo uses the *Rapid Action Value Estimate* (RAVE) algorithm in its transient memory (Gelly & Silver, 2007). This algorithm can also be viewed as a special case of the Dyna-2 architecture, but using features of the full history h_t and not just the current state s_t and action a_t.

We define a history to be a sequence of states and actions $h_t = s_1 a_1 ... s_t a_t$, including the current action a_t. An individual RAVE feature $\phi_{sa}^{RAVE}(h)$ is a binary feature of the history h that matches a particular state s and action a. The binary feature is on iff s occurs in the history and a matches the current action a_t,

$$\phi_{sa}^{RAVE}(s_1 a_1 ... s_t a_t) = \begin{cases} 1 & \text{if } a_t = a \text{ and } \exists i \text{ s.t. } s_i = s; \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

Thus the RAVE algorithm provides a simple abstraction over classes of related histories. The implementation of RAVE used in MoGo makes two additional simplifications. First, MoGo estimates a value for each RAVE feature independently of any other RAVE features, set to the average outcome of all simulations in which the RAVE feature ϕ_{sa}^{RAVE} is active. Second, for action selection, MoGo only evaluates the single RAVE feature $\phi_{s_t a_t}^{RAVE}$ corresponding to the current state s_t and candidate action a_t. This somewhat reduces the generalisation power of RAVE, but allows for a particularly efficient update procedure.

8. Conclusion

Reinforcement learning is often considered a slow procedure. Outstanding examples of success have, in the past, learned a value function from months of offline computation. However, this does not need to be the case. Many reinforcement learning methods are fast, incremental, and scalable. When such a reinforcement learning algorithm is applied to simulated experience, using a transient memory, it becomes a high performance search algorithm. This search procedure can be made more efficient by generalising across states; and it can be combined with long-term learning, using a permanent memory.

Monte-Carlo tree search algorithms, such as UCT, have recently received much attention. However, this

is just one example of a sample-based search algorithm. There is a spectrum of algorithms that vary from table-lookup to function approximation; from Monte-Carlo learning to bootstrapping; and from permanent to transient memories. Function approximation provides rapid generalisation in large domains; bootstrapping is advantageous in the presence of function approximation; and permanent and transient memories allow general knowledge about the past to be combined with specific knowledge about the expected future. By varying these dimensions, we have achieved a significant improvement over the UCT algorithm.

In 9×9 Go, programs based on extensions to the UCT algorithm have achieved *dan*-level performance. Our program RLGO, based on the Dyna-2 architecture, is the strongest program not based on UCT, and suggests that the full spectrum of sample-based search methods merits further investigation. For larger domains, such as 19×19 Go, generalising across states becomes increasingly important. Combining state abstraction with sample-based search is perhaps the most promising avenue for achieving human-level performance in this challenging domain.

References

Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time analysis of the multi-armed bandit problem. *Machine Learning*, *47*, 235–256.

Baxter, J., Tridgell, A., & Weaver, L. (1998). Experiments in parameter learning using temporal differences. *International Computer Chess Association Journal*, *21*, 84–99.

Buro, M. (1999). From simple features to sophisticated evaluation functions. *First International Conference on Computers and Games* (pp. 126–145).

Coulom, R. (2007). Computing Elo ratings of move patterns in the game of Go. *Computer Games Workshop*.

Dahl, F. (1999). Honte, a Go-playing program using neural nets. *Machines that learn to play games* (pp. 205–223). Nova Science.

Enzenberger, M. (2003). Evaluation in Go by a neural network using soft segmentation. *10th Advances in Computer Games Conference* (pp. 97–108).

Gelly, S., & Silver, D. (2007). Combining online and offline learning in UCT. *17th International Conference on Machine Learning* (pp. 273–280).

Gelly, S., Wang, Y., Munos, R., & Teytaud, O. (2006). *Modification of UCT with patterns in Monte-Carlo Go* (Technical Report 6062). INRIA.

Kocsis, L., & Szepesvari, C. (2006). Bandit based Monte-Carlo planning. *15th European Conference on Machine Learning* (pp. 282–293).

Paduraru, C. (2007). Planning with approximate and learned MDP models. Master's thesis, University of Alberta.

Rummery, G., & Niranjan, M. (1994). *On-line Q-learning using connectionist systems* (Technical Report CUED/F-INFENG/TR 166). Cambridge University Engineering Department.

Schaeffer, J., Hlynka, M., & Jussila, V. (2001). Temporal difference learning applied to a high-performance game-playing program. *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence* (pp. 529–534).

Schraudolph, N., Dayan, P., & Sejnowski, T. (1994). Temporal difference learning of position evaluation in the game of Go. *Advances in Neural Information Processing 6* (pp. 817–824).

Silver, D., Sutton, R., & Müller, M. (2007). Reinforcement learning of local shape in the game of Go. *20th International Joint Conference on Artificial Intelligence* (pp. 1053–1058).

Sutton, R. (1988). Learning to predict by the method of temporal differences. *Machine Learning*, *3*, 9–44.

Sutton, R. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. *7th International Conference on Machine Learning* (pp. 216–224).

Sutton, R. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in Neural Information Processing Systems 8* (pp. 1038–1044).

Sutton, R., & Barto, A. (1998). *Reinforcement learning: an introduction*. MIT Press.

Sutton, R., Koop, A., & Silver, D. (2007). On the role of tracking in stationary environments. *17th International Conference on Machine Learning* (pp. 871–878).

Tesauro, G., & Galperin, G. (1996). On-line policy improvement using Monte-Carlo search. *Advances in Neural Information Processing 9* (pp. 1068–1074).