Machine Learning manuscript No.
(will be inserted by the editor)

Temporal-Difference Search in Computer Go

David Silver[1], **Richard Sutton**[2] **and Martin Müller**[2]

Abstract *Temporal-difference learning* is one of the most successful and broadly applied solutions to the reinforcement learning problem; it has been used to achieve master-level play in chess, checkers and backgammon. The key idea is to update a value function from episodes of real experience, by bootstrapping from future value estimates, and using value function approximation to generalise between related states. *Monte-Carlo tree search* is a recent algorithm for high-performance search, which has been used to achieve master-level play in Go. The key idea is to use the mean outcome of simulated episodes of experience to evaluate each state in a search tree. We introduce a new approach to high-performance search in Markov decision processes and two-player games. Our method, *temporal-difference search*, combines temporal-difference learning with simulation-based search. Like Monte-Carlo tree search, the value function is updated from simulated experience; but like temporal-difference learning, it uses value function approximation and bootstrapping to efficiently generalise between related states. We apply temporal-difference search to the game of 9×9 Go, using a million binary features matching simple patterns of stones. Without any explicit search tree, our approach outperformed an unenhanced Monte-Carlo tree search with the same number of simulations. When combined with a simple alpha-beta search, our program also outperformed all traditional (pre-Monte-Carlo) search and machine learning programs on the 9×9 Computer Go Server.

1 Introduction

Reinforcement learning (RL) can be subdivided into two fundamental problems: *learning* and *planning*. The goal of learning is for an agent to improve its policy from its interactions with its environment. The goal of planning is for an agent to improve its policy without further interaction with its environment, by a process of deliberation during the available computation time.

Despite the apparent differences between these two problems, they are intimately related. During learning, the agent interacts with the real environment, by executing actions and observing their consequences. During planning the agent can interact with a *model* of the environment, by simulating actions and observing their consequences. In both cases the agent updates its policy from samples of experience. In this article we demonstrate that an

1. University College London 2. University of Alberta

agent can both learn *and* plan effectively, in large and challenging environments, by applying the method of temporal-differences to both real and simulated experience.

We focus on the game of Go as a concrete example of a large, challenging environment in which traditional approaches to learning and planning have failed to produce significant success. Go has more than 10^{170} states, up to 361 legal moves, and the long-term effect of a move may only be revealed after hundreds of additional moves. There are no simple heuristics by which to evaluate a position, and it has proven surprisingly difficult to encode human expert knowledge in machine usable form, either by handcrafted pattern databases or by machine learning (Müller, 2002).

Recently, a new family of search algorithms based on Monte-Carlo simulation, have revolutionised computer Go and achieved human master-level play for the first time (Coulom, 2006; Gelly and Silver, 2007). The key idea of these algorithms is to use the mean outcome of random simulations to evaluate positions. In *Monte-Carlo tree search* many thousands of random games are simulated by self-play, starting from the current position, and adding each new position into a search tree. The value of each position in the tree is estimated by the mean outcome of all simulations that visit that position. The search tree is used to guide simulations along the most promising paths, by selecting the child node with the highest value or potential value (Kocsis and Szepesvari, 2006).

However, Monte-Carlo tree search suffers from two major deficiencies. First, each position is evaluated independently, without any generalisation between similar positions. Second, Monte-Carlo simulation produces a high variance estimate of the value of each position. As a result, the basic Monte-Carlo tree search algorithm can require prohibitively large numbers of simulations when applied to very large search spaces, such as 19×19 Go. In practice, the strongest current Go programs deal with these issues by using domain specific prior knowledge (Chaslot et al., 2008), generalisation among subtrees (Gelly and Silver, 2011), and carefully hand-tuned simulation policies (Gelly et al., 2006). In this article we develop a much more general framework for simulation-based search that addresses these two weaknesses. This framework represents a spectrum of search algorithms that includes Monte-Carlo tree search, but also allows for two important dimensions of variation. First, it uses *value function approximation* to generalise between related positions. Second, it uses *bootstrapping* to reduce the variance of the estimated value.

Our framework draws extensively from successful examples of reinforcement learning in classic games such as chess (Veness et al., 2009), checkers (Schaeffer et al., 2001), and backgammon (Tesauro, 1994). In each of these games, *temporal-difference learning* (TD learning) has been used to achieve human master-level play. In each case, a value function was trained offline from games of self-play, typically requiring weeks or even months of computation. This value function was then used to evaluate leaf positions in a high-performance alpha-beta search.

However, in very large environments such as Go, it is difficult to construct a global value function with any degree of accuracy (Müller, 2002). Instead of weakly approximating the value of every position, we approximate the value of positions that occur in the subgame starting from *now* until termination. This new idea is implemented by re-training the value function *online*, by TD learning from games of self-play that start from the current position. This re-training procedure occurs in real-time, in a matter of seconds. The value function evolves dynamically throughout the course of the game, specialising more and more to the particular tactics and strategies that are relevant to *this* game and *this* position. We demonstrate that this method, which we call *temporal-difference search* (TD search), can provide a dramatic improvement to the quality of position evaluation.

In Section 3 we focus on a direct reinforcement learning approach to computer Go. This approach is based on a straightforward application of TD learning to 9×9 Go. We introduce a simple, expansive representation of Go knowledge, using *local shape features*, that requires no prior knowledge of the game except for the grid structure of the board. We present an empirical study of TD learning in 9×9 Go, using the program *RLGO*,[1] to develop intuitions about the core algorithms and concepts used throughout this article. Using TD learning and a million local shape features, RLGO achieved a competent level of play in 9×9 Go.

In Section 4 we focus on a simulation-based search approach to computer Go. In this section we develop our main idea: the TD search algorithm. We build on the reinforcement learning approach from Section 3, but here we apply TD learning *online* to the current subgame. We present an empirical study of TD search in 9×9 Go, again using local shape features in the program *RLGO*, to explore this new paradigm. Using TD search and a million local shape features, *RLGO* achieved a dramatic improvement of around 500 Elo points over TD learning. Without any explicit search tree, TD search achieved better performance than an unenhanced Monte-Carlo tree search.

The remainder of the article focuses on high-performance combinations of learning and search. In Section 5 we introduce the *Dyna-2* algorithm. This algorithm contains two sets of parameters: a long-term memory, updated by TD learning; and a short-term memory, updated by TD-search. Finally, in Section 6 we introduce a two-phase search that combines TD search with a traditional alpha-beta search (successfully) or a Monte-Carlo tree search (unsuccessfully). When both TD learning and TD search were combined together, using an enhanced alpha-beta search, *RLGO* achieved a higher rating on the 9×9 Computer Go Server than all traditional (pre-Monte-Carlo) search and machine learning programs.

2 Background

2.1 Markov Decision Processes and Markov Games

A Markov decision process (MDP) consists of a set of states $\mathcal{S}$ and a set of actions $\mathcal{A}$. The dynamics of the MDP, from any state s and for any action a, are determined by *transition probabilities*, $\mathcal{P}^a_{ss'} = Pr(s_{t+1} = s' | s_t = s, a_t = a)$, specifying the distribution over the next state s'. Finally, a *reward function*, $\mathcal{R}^a_{ss'} = \mathbb{E}[r_{t+1} | s_t = s, a_t = a, s_{t+1} = s']$, specifies the expected reward for a given state transition. We consider episodic MDPs which terminate with probability 1 in a distinguished terminal state. The *return* $R_t = \sum_{k=t}^{T} r_k$ is the total reward accumulated in that episode from time t until reaching the terminal state at time T.

A *policy*, $\pi(s, a) = Pr(a_t = a | s_t = s)$, maps a state s to a probability distribution over actions. The *value function*, $V^\pi(s) = \mathbb{E}_\pi[R_t | s_t = s]$, is the expected return from state s when following policy π. The *action value function*, $Q^\pi(s, a) = \mathbb{E}_\pi[R_t | s_t = s, a_t = a]$, is the expected return after selecting action a in state s and then following policy π. The *optimal value function* is the unique value function that maximises the value of every state, $V^*(s) = \max_\pi V^\pi(s), \forall s \in \mathcal{S}$ and $Q^*(s, a) = \max_\pi Q^\pi(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}$. An *optimal policy* $\pi^*(s, a)$ is a policy that maximises the action value function from every state in the MDP, $\pi^*(s, a) = \operatorname{argmax}_\pi Q^\pi(s, a)$.

A *symmetric alternating Markov game* (SAM game) is a natural generalisation of a Markov decision process to zero-sum, perfect information, two-player games such as chess, checkers, backgammon or Go (Littman, 1994). An *agent* and an *opponent* alternately select

[1] *RLGO* is open source software, `http://sourceforge.net/projects/rlgo`

actions from the same set, $a \in \mathcal{A}$, according to separate policies $\pi_1(s,a)$ and $\pi_2(s,a)$. As in an MDP, there are transition probabilities, $\mathcal{P}^a_{ss'}$, and a reward function, $\mathcal{R}^a_{ss'}$. The agent seeks to maximise its total reward, whereas the opponent seeks to minimise the total reward. A *self-play policy* is a single policy $\pi(s,a)$ that is used by both agent and opponent, $\pi_1(s,a) = \pi_2(s,a) = \pi(s,a)$. The value function, $V^\pi(s)$, is the expected return for the agent from state s when following self-play policy π. A *minimax optimal* policy for a SAM game is a self-play policy that both maximises the agent's value function and minimises the opponent's value function. This policy corresponds to optimal play under the worst-case assumption that the opponent plays perfectly.

2.2 Monte-Carlo Learning

Monte-Carlo evaluation provides a particularly simple, model-free method for estimating the value function for a given policy π. The value of state s is estimated by the mean return of all episodes that visit state s, without requiring any model of the MDP. This provides an unbiased, but high variance estimate of $V^\pi(s)$.

Monte-Carlo control (Sutton and Barto, 1998) combines Monte-Carlo evaluation with *ϵ–greedy policy improvement*. An action value function is estimated by Monte-Carlo evaluation, so that $Q(s,a)$ is the mean return of all episodes in which action a was selected from state s. An ϵ–greedy policy is used to combine exploration (selecting a random action with probability ϵ) with exploitation (selecting $\operatorname{argmax}_a Q(s,a)$ with probability $1-\epsilon$). If all states are visited infinitely many times, and ϵ decays to zero in the limit, Monte-Carlo control converges to the optimal action-value function $Q^*(s,a)$ for all states s and actions a (Tsitsiklis, 2002), and hence to the optimal policy $\pi^*(s,a)$.

2.3 Temporal-Difference Learning

Temporal-difference learning (TD learning) is a model-free method for policy evaluation that bootstraps the value function from subsequent estimates of the value function. In the *TD(0)* algorithm, the value function is bootstrapped from the very next time-step. Rather than waiting until the complete return has been observed, the value function of the next state is used to approximate the expected return. The *TD-error* δV_t is measured between the value at state s_t and the value at the subsequent state s_{t+1}. For example, if the agent thinks that Black is winning in position s_t, but that White is winning in the next position s_{t+1}, then this inconsistency generates a TD-error. The TD(0) algorithm adjusts the value function so as to correct the TD-error and make it more consistent with the subsequent value,

$$\begin{aligned} \delta V_t &= r_{t+1} + V(s_{t+1}) - V(s_t) \\ \Delta V(s_t) &= \alpha \delta V_t \end{aligned} \tag{1}$$

where α is a step-size parameter controlling the learning rate.

The *TD(λ)* algorithm bootstraps the value of a state from subsequent values many steps into the future. The parameter $\lambda \in [0,1]$ determines the temporal span over which bootstrapping occurs. At one extreme, TD(0) bootstraps the value of a state only 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.

To implement TD(λ) online, an eligibility trace $e(s)$ is maintained for each state. The eligibility trace represents the total credit assigned to a state for any subsequent errors in evaluation. It combines a *recency heuristic* with a *frequency heuristic*: states which are visited most frequently and most recently are given the greatest eligibility (Sutton, 1984),

$$\begin{aligned} e_t(s) &= \begin{cases} \lambda e_{t-1}(s) + 1 & \text{if } s = s_t \\ \lambda e_{t-1}(s) & \text{otherwise} \end{cases} \\ \Delta V_t(s) &= \alpha \delta V_t e_t(s) \end{aligned} \tag{2}$$

If all states are visited infinitely many times, and with appropriate step-sizes, TD(λ) converges to V^π for all values of λ (Dayan and Sejnowski, 1994).

The *Sarsa(λ)* algorithm (Rummery and Niranjan, 1994) combines TD(λ) with ϵ–greedy policy improvement. Sarsa estimates an action value function, $Q(s, a)$, for each state s and action a. At each time-step t, the next action a_t is selected by an ϵ–greedy policy with respect to the current action values $Q(s, \cdot)$. The action value function is then updated from the sample tuple of experience, $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$, by using the TD($\lambda$) update rule for action values,

$$\begin{aligned} \delta Q_t &= r_{t+1} + Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \\ e_t(s, a) &= \begin{cases} \lambda e_{t-1}(s, a) + 1 & \text{if } s = s_t \text{ and } a = a_t \\ \lambda e_{t-1}(s, a) & \text{otherwise} \end{cases} \\ \Delta Q(s, a) &= \alpha \delta Q_t e_t(s, a) \end{aligned} \tag{3}$$

If all states are visited infinitely many times, and ϵ decays to zero in the limit, Sarsa(λ) converges to the optimal action-value function Q^* for all values of λ (Singh et al., 2000).

TD learning can be applied to SAM games in an analogous fashion. The *self-play Sarsa(λ)* algorithm estimates an action value function $Q(s, a)$ for the current self-play policy, again updated by TD(λ). The next action a_t is selected by an ϵ–greedy self-play policy that selects, with probability $1 - \epsilon$, the action with maximum value for the agent, or the action with minimum value for the opponent. The convergence results for TD learning can be extended directly to SAM games (Littman, 1996). Under similar conditions self-play Sarsa will converge on the minimax optimal value function.

In many games and environments, it can be more efficient to consider the *afterstate value* $V(s \circ a)$, for the "afterstate" reached after selecting action a from state s, rather than the action-value $Q(s, a)$ (Sutton and Barto, 1998). The *afterstate Sarsa* algorithm is identical to Sarsa(λ) except that it applies TD(λ) to afterstate values instead of action values. In deterministic problems, the afterstate is simply the successor state, and the afterstate Sarsa(λ) update is equivalent to the basic TD(λ) update in Equation 2.

2.4 Value Function Approximation

In *table lookup* the value function has a distinct value $V(s)$ for each state s. In large environments table lookup is not practical and the value function must be approximated. A common and successful approach (Sutton, 1996) is to use a feature vector $\phi(s)$ to represent state s, and to approximate the value function $V(s)$ by a linear combination of features $\phi(s)$ and weights θ.

The *linear TD(0)* algorithm combines the TD(0) algorithm with linear value function approximation. Each parameter is updated in proportion to the TD-error and the corresponding feature value. It is analogous to stochastic gradient descent algorithms for linear regression, such as the LMS or Widrow-Hoff rule (Widrow and Stearns, 1985),

$$V(s) = \phi(s) \cdot \theta \tag{4}$$
$$\Delta\theta = \alpha\delta V_t \phi(s_t) \tag{5}$$

Many problems are best described by a binary outcome z indicating success or failure, *e.g.* winning a game, solving a puzzle or achieving a goal state. These problems can be described by a terminal reward of $r = 1$ for success, $r = 0$ for failure, and no intermediate rewards. In this case, the value function $V^\pi(s)$ is equal to the probability of achieving a successful outcome from state s under policy π, $V^\pi(s) = Pr(z = 1|s, \pi)$. However, linear approximation is not well suited to modelling probabilities (Jordan, 1995). Instead, a logistic function, $\sigma(x) = \frac{1}{1+e^{-x}}$, can be used to squash the value function into the desired range $[0, 1]$. The *logistic TD(0)* algorithm (Silver, 2009) is a temporal-difference learning algorithm for logistic-linear value function approximation. It minimises the cross-entropy between the current and subsequent value estimates, and is analogous to stochastic gradient descent algorithms for logistic regression (Jordan, 1995),

$$V(s) = \sigma(\phi(s) \cdot \theta)$$
$$\Delta\theta = \alpha\delta V_t \phi(s_t) \tag{6}$$

The *linear TD(λ)* algorithm combines TD(λ) with linear value function approximation. Similarly, the *logistic TD(λ)* algorithm combines TD(λ) with logistic-linear value function approximation. In both cases, an eligibility trace vector e represents the credit assigned to each feature for subsequent errors,

$$e_t = \lambda e_{t-1} + \phi(s_t) \tag{7}$$
$$\Delta\theta = \alpha\delta V_t e_t \tag{8}$$

The *linear Sarsa(λ)* algorithm combines linear TD(λ) with ϵ–greedy policy improvement (see Algorithm 1). The action value function is approximated by a linear combination of features, $Q(s, a) = \phi(s, a) \cdot \theta$, where $\phi(s, a)$ is now a feature vector representing both state s and action a. Although there are no guarantees of convergence, linear Sarsa chatters within some bounded region of the optimal value function (Gordon, 1996). The *logistic Sarsa(λ)* algorithm is identical except that it uses logistic linear value function approximation, $Q(s, a) = \sigma(\phi(s, a) \cdot \theta)$.

3 Temporal-Difference Learning with Local Shape Features

In this section we learn a position evaluation function for the game of Go, without requiring any domain knowledge beyond the grid structure of the board. We use a simple representation, based on local 1×1 to 3×3 patterns, that is intended to capture intuitive shape knowledge, and can be computed particularly efficiently in the context of a high-performance search. We evaluate positions using a linear combination of these pattern features, and learn weights by TD learning and self-play. This same approach could in principle be used to automatically construct an evaluation function for many other games.

Algorithm 1 Linear Sarsa(λ)

procedure LINEAR-SARSA(λ)
  $\theta \leftarrow 0$ ▷ Clear weights
  loop
    $s \leftarrow s_0$ ▷ Start new episode in initial state
    $e \leftarrow 0$ ▷ Clear eligibility trace
    $a \leftarrow \epsilon$–greedy action from state s
    while s is not terminal **do**
      Execute a, observe reward r and next state s'
      $a' \leftarrow \epsilon$–greedy action from state s'
      $\delta Q \leftarrow r + Q(s', a') - Q(s, a)$ ▷ Calculate TD-error
      $\theta \leftarrow \theta + \alpha \delta Q e$ ▷ Update weights
      $e \leftarrow \lambda e + \phi(s, a)$ ▷ Update eligibility trace
      $s \leftarrow s', a \leftarrow a'$
    end while
  end loop
end procedure

3.1 Shape Knowledge in Go

The concept of shape is extremely important in Go. A good shape uses local stones efficiently to maximise tactical advantage (Matthews, 2003). Professional players analyse positions using a large vocabulary of shapes, such as *joseki* (corner patterns) and *tesuji* (tactical patterns). The joseki at the bottom left of Figure 1a is specific to the white stone on the 4-4 intersection,[2] whereas the tesuji at the top-right could be used at any location. Shape knowledge may be represented at different scales, with more specific shapes able to specialise the knowledge provided by more general shapes (Figure 1b). Many Go proverbs exist to describe shape knowledge, for example "*ponnuki* is worth 30 points", "the one-point jump is never bad" and "*hane* at the head of two stones" (Figure 1c).

3.1.1 Prior Approaches to Shape Knowledge Representation

Commercial computer Go programs rely heavily on the use of pattern databases to represent shape knowledge (Müller, 2002). Many man-years have been devoted to hand-encoding professional expertise in the form of local pattern rules. Each pattern recommends a move to be played whenever a specific configuration of stones is encountered on the board. The configuration can also include additional features, such as requirements on the liberties or strength of a particular stone. Unfortunately, pattern databases suffer from the knowledge acquisition bottleneck: expert shape knowledge is hard to quantify and encode, and the interactions between different patterns may lead to unpredictable behaviour.

Prior work on learning shape knowledge has focused on predicting expert moves by supervised learning (Stoutamire, 1991; van der Werf et al., 2002; Stern et al., 2006). This approach has achieved a 30–40% success rate in predicting the move selected by a human player, across a large data-set of human expert games. However, it has not led directly to strong play in practice, perhaps due to its focus on mimicking rather than understanding a position. For example, supervised learning programs often respond appropriately to familiar moves, but respond bizarrely to unusual moves.

A second approach has been to train a multi-layer perceptron, using TD learning by self-play (Schraudolph et al., 1994). The networks implicitly contain some representation

[2] Intersections are indexed inwards from the corners, starting at 1-1 for the corner intersection itself.

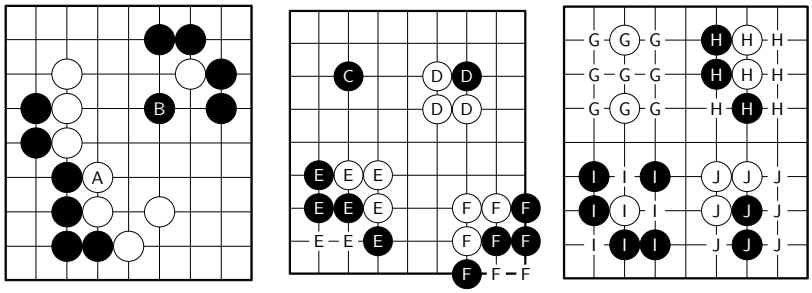


Fig. 1 a) The pattern of stones near A forms a common *joseki* that is specific to the 4-4 intersection. Black B captures the white stone using a *tesuji* that can occur at any location. b) In general a stone on the 3-3 intersection (C) helps secure a corner. If it is surrounded then the corner is insecure (D), although with sufficient support it will survive (E). However, the same shape closer to the corner is unlikely to survive (F). c) Go players describe positions using a large vocabulary of shapes, such as the *one-point jump* (G), *hane* (H), *net* (I) and *turn* (J).

of local shape, and utilise weight sharing to exploit the natural symmetries of the Go board. This approach has led to stronger Go playing programs, such as Dahl's *Honte* (Dahl, 1999) and Enzenberger's *NeuroGo* (Enzenberger, 2003). However, these networks utilise a great deal of specific Go knowledge in the network architecture and input features. Furthermore, the computational cost of these networks is relatively high, reducing their desirability for the high-performance search algorithms discussed in the remainder of this article.

3.1.2 Local Shape Features

Local shape features are a simple representation of shape knowledge that is particularly fast to compute. They represent the set of local board configurations within all square regions up to size $m \times m$.

A state in the game of Go, $s \in \{\cdot, \circ, \bullet\}^{N \times N}$, consists of a state variable for each intersection of a size $N \times N$ board, with three possible values for empty, black and white stones respectively.[3] We define a *local shape* l to be a specific configuration of state variables within some size k square region of the board, $l \in \{\cdot, \circ, \bullet\}^{k \times k}$. We exhaustively enumerate all possible local shapes within all square regions up to size $k \leq m$, and at all possible locations (including overlapping locations). The *local shape feature* $\phi_i(s)$ has value 1 in state s if the board exactly matches the ith local shape l_i, and value 0 otherwise.

The local shape features are combined into a large feature vector $\phi(s)$. This feature vector is very sparse: exactly one local shape is matched in each square region of the board; all other local shape features have value 0.

3.1.3 Weight Sharing

We use weight sharing to exploit the symmetries of the Go board (Schraudolph et al., 1994). We define an equivalence relationship over local shapes, such that all rotationally and reflectionally symmetric local shapes are placed in the same equivalence class. In addition, each equivalence class includes *inversions* of the local shape, in which all black and white stones are flipped to the opposite colour.

[3] Technically the state also includes the full board history, so as to avoid repetitions (known as *ko*).

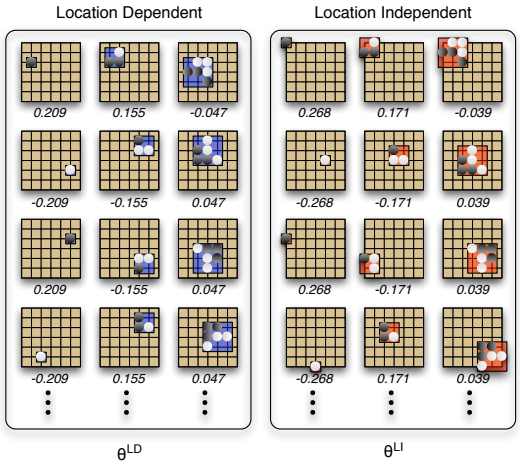


Fig. 2 Examples of location dependent and location independent weight sharing.

The local shape with the smallest index j within the equivalence class is considered to be the canonical example of that class. Every local shape feature ϕ_i in the equivalence class shares the weight θ_j of the canonical example, but the sign may differ. If the local shape feature has been inverted from the canonical example, then it uses negative weight sharing, $\theta_i = -\theta_j$, otherwise it uses positive weight sharing, $\theta_i = \theta_j$. In certain equivalence classes (for example empty local shapes), an inverted shape is identical to an uninverted shape, so that either positive or negative weight sharing could be used, $\theta_i = \theta_j = -\theta_j \Rightarrow \theta_i = \theta_j = 0$. We describe these local shapes as *neutral*, and assume that they are equally advantageous to both sides. All neutral local shapes are removed from the representation.

Rotational and reflectional symmetries are exploited by applying a whole-board rotation and/or reflection to a local shape. The vector of *location dependent* weights θ^{LD} shares weights among all rotational, reflectional and colour inversion symmetries. The vector of *location independent* weights θ^{LI} also incorporates translation symmetry: all local shapes that have the same configuration, regardless of its absolute location on the board, are included in the same equivalence class. Figure 2 shows some examples of both types of weight sharing.

For each size of square up to 3×3, all local shape features are exhaustively enumerated, using both location dependent and location independent weights. This provides a hierarchy of local shape features, from very general configurations that occur many times each game, to specific configurations that are rarely seen in actual play. Smaller local shape features are more general than larger ones, and location independent weights are more general than location dependent weights. The more general features and weights provide no additional information, but may offer a useful abstraction for rapid learning. Table 1 shows, for 9×9 Go, the total number of local shape features of each size; the total number of distinct equivalence classes, and therefore the total number of unique weights; and the maximum number of active features (features with value of 1) in the feature vector.

Local shape features	Total features	Unique weights	Max active features
1×1 Location Independent	243	1	81
1×1 Location Dependent		15	81
2×2 Location Independent	5184	8	64
2×2 Location Dependent		344	64
3×3 Location Independent	964467	1418	49
3×3 Location Dependent		61876	49
Total	969894	63303	388

Table 1 Number of local shape features of different sizes in 9×9 Go. The number of unique weights describes the number of distinct equivalence classes using weight sharing.

3.2 Learning Algorithm

Our objective is to win games of Go. This goal can be expressed by a binary reward function, which gives a reward of $r = 1$ if Black wins and $r = 0$ if White wins, with no intermediate rewards. The value function $V^\pi(s)$ is defined to be the expected total reward from state s when following policy π. This value function is Black's *winning probability* from state s (see Section 2.4). Black seeks to maximise his winning probability, while White seeks to minimise it. We approximate the value function by a logistic-linear combination of local shape features and both location dependent and location independent weights,

$$V(s) = \sigma\left(\phi(s) \cdot \theta^{LI} + \phi(s) \cdot \theta^{LD}\right) \tag{9}$$

We measure the TD-error between the current value $V(s_t)$, and the value after both player and opponent have made a move, $V(s_{t+2})$. In this approach, which we refer to as a *two–ply update*, the value is updated between successive moves with the same colour to play. The current player is viewed as the agent, and his opponent is viewed as part of the environment. We contrast this approach to a *one–ply update*, used in prior work such as *TD-Gammon* (Tesauro, 1994) and *NeuroGo* (Enzenberger, 2003), that measures the TD-error between Black and White moves.

We update both location dependent and location independent weights by logistic TD(0) (see Section 2.4). For each feature ϕ_i, the shared value for the corresponding weights θ_i^{LI} and θ_i^{LD} is updated. This can lead to the same shared weight being updated many times in a single time-step.[4]

It is well-known that TD learning, much like the LMS algorithm in supervised learning, is sensitive to the choice of learning rate (Singh and Dayan, 1998). If the features are scaled up or down in value, or if more or less features are included in the feature vector, then the learning rate needs to change accordingly. To address this issue, we divide the step-size by the total number of currently active features, $||\phi(s_t)||^2 = \sum_{i=1}^{n} \phi(s_t)^2$. As in the normalised LMS algorithm, (Haykin, 1996), this normalises the update by the total signal power of the features. We refer to this normalised, two–ply, logistic TD(0) update as *logistic NTD2(0)*,

$$\Delta\theta^{LD} = \Delta\theta^{LI} = \alpha \frac{\phi(s_t)}{||\phi(s_t)||^2}(V(s_{t+2}) - V(s_t)) \tag{10}$$

Our control algorithm is based on self-play, afterstate Sarsa (see Section 2.3), using the logistic NTD2(0) update. The policy is updated after every move t, by using an ϵ–greedy

[4] An equivalent state representation would be to have one feature $\phi_i(s)$ for each equivalence class i, where $\phi_i(s)$ counts the number of occurrences of equivalence class i in state s.

policy. With probability $1 - \epsilon$ the player selects the move a that maximises (Black) or minimises (White) the afterstate value $V(s \circ a)$. With probability ϵ the player selects a move with uniform random probability. The learning update is applied whether or not an exploratory move is selected.

Because the local shape features are sparse, only a small subset of features need be evaluated and updated. This leads to an an efficient $O(k)$ implementation, where k is the total number of active features. This requires just a few hundred operations, rather than evaluating or updating a million components of the full feature vector.

3.2.1 Training Procedure

We initialise all weights to zero, so that rarely encountered features do not initially contribute to the evaluation. We train the weights by executing a million games of self-play in 9×9 Go. Both Black and White select moves using an ϵ–greedy policy over the same value function $V(s)$. The same weights are used by both players, and are updated after both Black and White moves by logistic NTD2(0).

All games begin from the empty board position, and terminate when both players pass. To prevent games from continuing for an excessive number of moves, we prohibit moves within single-point eyes, and only allow the pass move when no other legal moves are available. Games are scored by Chinese rules, assuming that all stones are alive, and using a *komi* of 7.5.

3.2.2 Computational Performance

On a 2 Ghz processor, using the default parameters, *RLGO 1.0* evaluates approximately 500,000 positions per second. *RLGO* uses a number of algorithmic optimisations in order to efficiently and incrementally update the value function. Nevertheless, the dominant computational cost in *RLGO* is position evaluation during ϵ–greedy move selection. In comparison, the logistic NTD2(0) learning update is relatively inexpensive, and consumes less than 10% of the overall computation time.

3.3 A Case Study of TD Learning in 9×9 Computer Go

In this empirical study we trained *RLGO 1.0* for a million games of self-play, with each of several different parameter settings, and saved the weights θ at logarithmically spaced intervals. We then ran a tournament between multiple instances of *RLGO*, each using a different vector of saved weights. In addition, we included the well-known baseline program *GnuGo* 3.7.10 (level 10), a strong traditional search program, in every tournament. Each tournament consisted of at least 1,000 games for each instance of *RLGO*.

After all matches were complete, the results were analysed by the *bayeselo* program to establish an *Elo* rating for every program. The Elo scale (Elo, 1978) assumes a logistic distribution with winning probability $Pr(A \text{ beats } B) = \frac{1}{1+10^{\frac{\mu_B - \mu_A}{400}}}$, where μ_A and μ_B are the Elo ratings for player A and player B respectively. On this scale, a difference of 200 Elo corresponds to a ~75% winning rate for the stronger player, and a difference of 500 Elo corresponds to a ~95% winning rate. Following convention, *GnuGo* (level 10) anchors the scale with a constant rating of 1800 Elo. Each plot contains error bars corresponding to 95% confidence intervals over these Elo ratings.

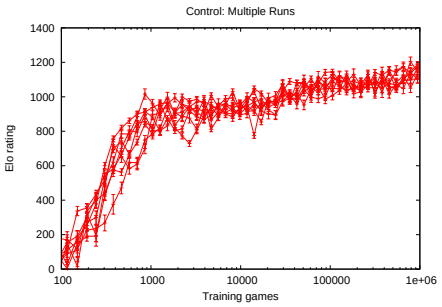


Fig. 3 Multiple runs using the default learning algorithm and local shape features.

Unless otherwise specified, we used the NTD2(0) learning algorithm (Equation 10) with logistic-linear value function approximation (Equation 9) and default parameter settings of $\alpha = 0.1$ and $\epsilon = 0.1$. All local shape features were used for all square regions from 1×1 up to 3×3, using both location dependent and location independent weight sharing. During tournament testing games, moves were selected by a simple one–ply maximisation (Black) or minimisation (White) of the value function, with no random exploration.

Figure 3 demonstrates that our learning algorithm is fairly consistent, producing similar performance over the same timescale in 8 different training runs with the same parameters, with a performance ranging over ~100 Elo at the end of training. It is therefore reasonable to assume that performance differences in excess of 100 Elo between individual training runs, are likely to be significant. Due to computational constraints the remaining experiments are based on a single training run for each parameter setting.

3.3.1 Local Shape Features in 9×9 *Go*

Perhaps the single most important property of local shape features is their huge range of generality. To assess this range, we counted the number of times that each equivalence class of feature occurs during training, and plotted a histogram for each size of local shape and for each type of weight sharing (see Figure 4). Each histogram forms a characteristic curve in log-log space. The most general class, the location independent 1×1 feature representing the material value of a stone, occurred billions of times during training. At the other end of the spectrum, there were tens of thousands of location dependent 3×3 features that occurred just a few thousand times, and 2,000 that were never seen at all. In total, each class of feature occurred approximately the same amount overall, but these occurrences were distributed in very different ways. Our learning algorithm must cope with this varied data: high-powered signals from small numbers of general features, and low-powered signals from a large number of specific features.

We ran several experiments to analyse how different combinations of local shape features affect the learning rate and performance of *RLGO 1.0*. In our first experiment, we used a single size of square region (see Figure 5(a)). The 1×1 local shape features, unsurprisingly, performed poorly. The 2×2 local shape features learnt very rapidly, but their representational capacity was saturated at around 1000 Elo after approximately 2,000 training games. The 3×3 local shape features learnt very slowly, but exceeded the performance of the 2×2 features after around 100,000 training games.

In our next experiment, we combined multiple sizes of square region (see Figures 5(b) and 5(c)). Using all features up to 3×3 effectively combined the rapid learning of the 2×2

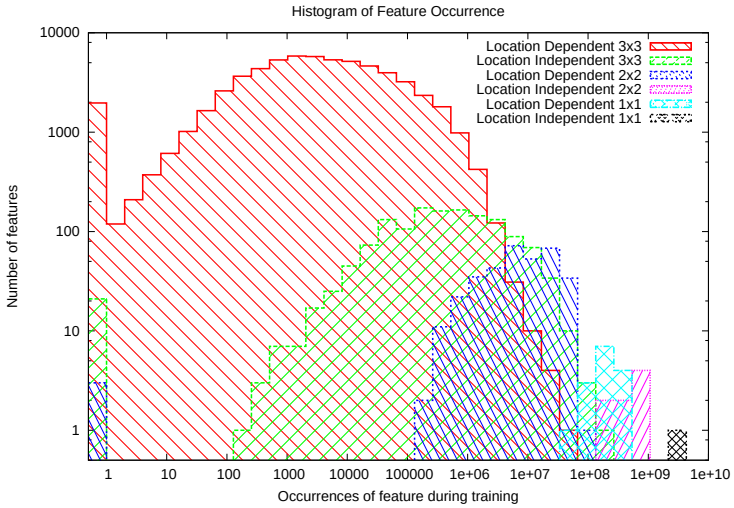


Fig. 4 Histogram of feature occurrences during a training run of 1 million games.

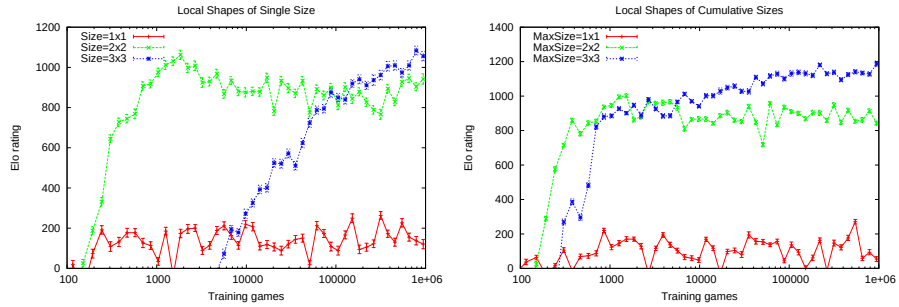
features with the better representational capacity of the 3×3 features; the final performance was better than for any single shape set, reaching 1200 Elo, and apparently still improving slowly. In comparison, the 3×3 features alone learnt much more slowly at first, taking more than ten times longer to reach 1100 Elo We conclude that a redundant representation, in which the same information is represented at multiple levels of generality, confers a significant advantage for at least a million training games.

We also compared a variety of different weight sharing schemes (see Figure 5(d)). Without any weight sharing, learning was very slow, eventually achieving 1000 Elo after a million training games. Location dependent weight sharing provided an intermediate rate of learning, and location independent weights provided the fastest learning. The eventual performance of the location independent weights was equivalent to the location dependent weights, and combining both types of weight sharing together offered no additional benefits. This suggests that the additional knowledge offered by location dependent shapes, for example patterns that are specific to edge or corner situations, was either not useful or not successfully learnt within the training time of these experiments.

3.3.2 Logistic Temporal-Difference Learning

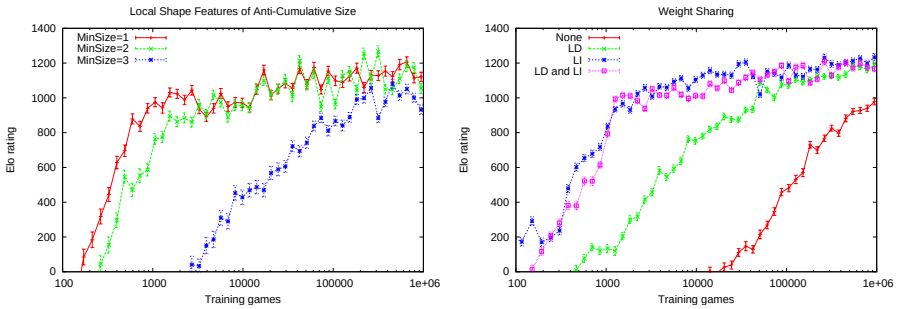
In Figures 6(a) and 6(b) we compare our logistic TD learning algorithm to a linear TD learning algorithm, for a variety of different step-sizes α. In the latter approach, the value function is represented directly by a linear combination of features, with no logistic function; the weight update equation is otherwise identical to logistic NTD2(0) (Equation 10).

Logistic TD learning is considerably more robust to the choice of step-size. It achieved good performance across three orders of magnitude of step-size, and improved particularly quickly with an aggressive learning rate. With a large step-size, the value function steps up or down the logistic function in giant strides. This effect can be visualised by zooming out of the logistic function until it looks much like a step function. In contrast, linear TD learning was very sensitive to the choice of step-size, and diverged when the step-size was too large, $\alpha \geq 0.1$.



(a) Learning curves for one size of local shape feature: 1×1, 2×2 and 3×3

(b) Cumulative sizes: 1×1; 1×1 and 2×2; and 1×1, 2×2 and 3×3



(c) Anti-cumulative sizes: 1×1, 2×2 and 3×3; 2×2 and 3×3; and 3×3

(d) Learning curves for different weight sharing rules

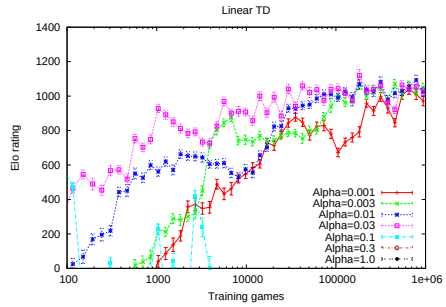
Fig. 5 TD learning in 9×9 Go with different sets of local shape features.

Logistic TD learning also achieved better eventual performance. This suggests that, much like logistic regression for supervised learning (Jordan, 1995), the logistic representation is better suited to representing probabilistic value functions.

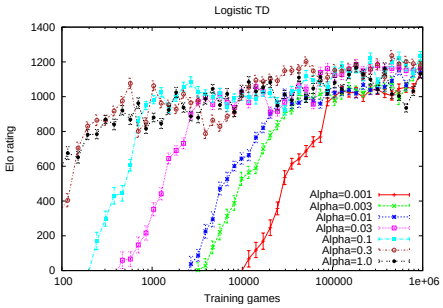
3.3.3 One and two–ply Updates

When training from self-play, TD learning can use either one–ply or two–ply updates (see Section 3.2). We compare the performance of these two updates in Figure 6(c). Surprisingly, one–ply updates, which were so effective in *TD-Gammon*, performed very poorly in *RLGO*. This is due to our more simplistic representation: *RLGO* does not differentiate between Black's turn and White's turn to play. As a result, whenever a player places down a stone, the value function is improved for that player. This leads to a large TD-error corresponding to the current player's advantage, which cannot ever be corrected. This error signal overwhelms the information about the relative strength of the move, compared to other possible moves. By using two–ply updates, this problem can be avoided altogether.[5] An alternative approach would be to use a richer representation, so that identical positions can be differentiated depending on the colour to play, as was done in *TD-Gammon*.

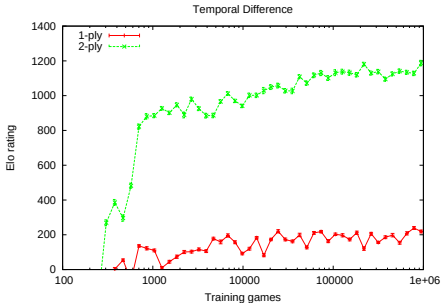
[5] Mayer also reports an advantage to two–ply TD(0) when using a simple multi-layer perceptron (Mayer, 2007).



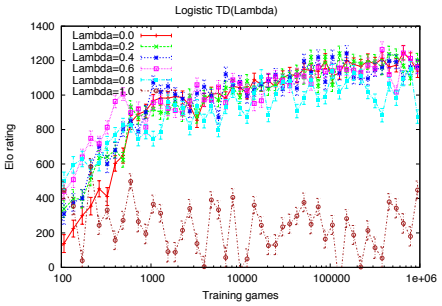
(a) Linear NTD2(0). Weights diverged for step-sizes of $\alpha \geq 0.1$



(b) Logistic NTD2(0).



(c) Learning curves for one–ply and two–ply updates.



(d) Logistic NTD2(λ) with different values of λ.

Fig. 6 Parameter study for TD learning in 9×9 Go.

3.3.4 Logistic TD(λ)

The logistic TD learning algorithm can incorporate a λ parameter that determines the time-span of the temporal difference (see Section 2.4). Here, we initialised all eligibility traces to zero at the start of each game, and used a normalised, two–ply variant of logistic TD(λ) that we refer to as NTD2(λ),

$$e_{t+1} \leftarrow \lambda e_t + \frac{\phi(s_t)}{||\phi(s_t)||^2} \tag{11}$$

$$\Delta\theta_t = \alpha(V(s_{t+2}) - V(s_t))e_t \tag{12}$$

In Figure 6(d) we compare the performance of logistic NTD2(λ) for different settings of λ. High values of λ, especially $\lambda = 1$, performed substantially worse, presumably due to the higher variance of the updates. The difference between lower values of λ was not significant.

3.3.5 Alpha-Beta Search

To complete our study of position evaluation in 9×9 Go, we used the learnt value function $V(s)$ as a heuristic function to evaluate the leaf positions in a fixed-depth alpha-beta search. We ran a tournament between several versions of *RLGO 1.0*, including *GnuGo* as a

Search depth	Elo rating	Standard Error
Depth 1	859	± 23
Depth 2	1067	± 20
Depth 3	1229	± 18
Depth 4	1226	± 20
Depth 5	1519	± 19
Depth 6	1537	± 19

Table 2 Performance of full width, fixed depth, alpha-beta search, using the learnt weights as an evaluation function. Weights were trained using default settings for 1 million training games. Elo ratings were established by a tournament amongst several players using the same weights. Each player selected moves by an alpha-beta search of the specified depth.

Search depth	**Elo rating on CGOS**
1	1050
5	1350

Table 3 Elo ratings established by *RLGO 1.0* on the first version of the Computer Go Server, using 10 minute time controls against a variety of computer opponents.

benchmark player, using an alpha-beta search of various fixed depths; the results are shown in Table 2.

Tournaments between alpha-beta search engines based on the same evaluation function often exaggerate the performance benefit of additional search. To gain some additional insight into the performance of our program, *RLGO* played online in tournament conditions against a variety of different opponents on the Computer Go Server (CGOS). The Elo rating established by *RLGO 1.0* is shown in Table 3.

The approach used by *RLGO* represents a departure from the search methods used in many previous computer Go programs. Traditional search programs such as *The Many Faces of Go* versions 1–11, and *GnuGo* favour a heavyweight, knowledge intensive evaluation function, which can typically evaluate a few hundred positions with a shallow global search. In contrast, *RLGO 1.0* combines a fast, lightweight evaluation function with a deeper, global search that evaluates millions of positions. Using a naive, fixed-depth alpha-beta search, *RLGO 1.0* was not able to compete with the heavyweight knowledge used in previous approaches. However, a fast, simple evaluation function can be exploited in many ways, and we explore some of these possibilities in the remainder of this article. In the next section we view temporal-difference learning as an essential element of the search algorithm, rather than an offline preprocessing step that is completed before search begins.

4 Temporal-Difference Search

In a SAM game $\mathcal{G}$, the current state s_t defines a new game, $\mathcal{G}'_t$, that is specific to this state. In the subgame $\mathcal{G}'_t$, the rules are the same, but the game always starts from state s_t. It may be substantially easier to solve or perform well in the subgame $\mathcal{G}'_t$, than to solve or perform well in the original game $\mathcal{G}$: the search space is reduced and a much smaller class of positions will be encountered. The subgame can have very different properties to the original game: certain patterns or features will be successful in this particular situation, which may not in general be a good idea. The idea of *temporal-difference search* (TD search) is to apply TD learning to $\mathcal{G}'_t$, using subgames of self-play that start from the current state s_t.

TD search can also be applied to any MDP $\mathcal{M}$. The current state s_t defines a sub-MDP ${\mathcal{M}'}_t$ that is identical to $\mathcal{M}$ except that the initial state is s_t. Again, the local sub-MDP ${\mathcal{M}'}_t$

may be much easier to solve or approximate than the full MDP $\mathcal{M}$. TD search applies TD learning to ${\mathcal{M}'}_t$, by generating episodes of experience that start from the current state s_t.

Rather than trying to learn a policy that covers every possible eventuality, TD search focuses on the subproblem that arises from the current state: how to perform well *now*. Life is full of such situations: you don't need to know how to climb every mountain in the world; but you'd better have a good plan for the one you are scaling right now. We refer to this idea of focusing the agent's resources on the current moment as *temporality*.

4.1 Simulation-Based Search

In reinforcement learning, the agent samples episodes of real experience and updates its value function from real experience. In *simulation-based search* the agent samples episodes of simulated experience and updates its value function from simulated experience. This symmetry between learning and planning has an important consequence: algorithms for reinforcement learning can also become algorithms for planning, simply by substituting simulated experience in place of real experience.

Simulation-based search requires a *generative model* of the MDP or SAM game, which can sample state transitions and rewards from $\mathcal{P}^a_{ss'}$ and $\mathcal{R}^a_{ss'}$ respectively. However, it is not necessary to know these probability distributions explicitly; the next state and reward could be generated by a black box simulator. The effectiveness of simulation-based search depends in large part on the accuracy of the model. In this article we sidestep the model learning problem and only consider MDPs or SAM games for which we have a perfect generative model or perfect knowledge of the game rules.

Simulation-based search algorithms sample experience in sequential episodes. Each simulation begins in a root state s_0. At each step u of simulation, an action a_u is selected according to a *simulation policy*, and a new state s_{u+1} and reward r_{u+1} is generated by the MDP model. This process repeats, without backtracking, until a terminal state is reached. The values of states or actions are then updated from the simulated experience.

Simulation-based search is usually applied online, at every time-step t, by initiating a new search that starts from the current state $s_0 = s_t$. The distribution of simulations then represents a probability distribution over future experience from time-step t onwards. Simulation-based search exploits temporality by learning from this specific distribution of future experience, rather than learning from the distribution of all possible experience.[6] Furthermore, as the agent's policy improves, the future experience distribution will become more refined. It can focus its value function on what is likely to happen next, given the improved policy, rather than learning about all possible eventualities.

4.1.1 Monte-Carlo Search

Monte-Carlo simulation is a very simple simulation-based search algorithm for evaluating candidate actions from a root state s_0. The search proceeds by simulating complete episodes from s_0 until termination, using a fixed simulation policy. The action values $Q(s_0, a)$ are estimated by the mean outcome of all simulations with candidate action a.

[6] In non-ergodic environments, such as episodic tasks, this distribution can be very different. However, even in ergodic environments, the short-term distribution of experience, generated by discounting or by truncating the simulations after a small number of steps, can be very different from the stationary distribution. This local *transient* in the problem can be exploited by an appropriately specialised policy.

Monte-Carlo tree search (MCTS) is perhaps the best-known example of a simulation-based search algorithm. It makes use of Monte-Carlo simulation to evaluate the nodes of a search tree (Coulom, 2006). There is one node, $n(s)$, corresponding to every state s encountered during simulation. Each node contains a total count for the state, $N(s)$, and a value $Q(s,a)$ and count $N(s,a)$ for each action $a \in \mathcal{A}$. Simulations start from the root state s_0, and are divided into two stages. When state s_u is contained in the search tree, a *tree policy* selects the action with the highest value or highest potential value. Otherwise, a random *default policy* is used to roll out simulations to completion. After each simulation $s_0, a_0, r_1, s_1, a_1, ..., r_T$, each node $n(s_u)$ in the search tree is updated incrementally to maintain the count and mean return from that node,

$$N(s_u) \leftarrow N(s_u) + 1 \tag{13}$$

$$N(s_u, a_u) \leftarrow N(s_u, a_u) + 1 \tag{14}$$

$$Q(s_u, a_u) \leftarrow Q(s_u, a_u) + \frac{R_u - Q(s_u, a_u)}{N(s_u, a_u)} \tag{15}$$

The *Upper Confidence Tree* (UCT) algorithm (Kocsis and Szepesvari, 2006) is a Monte-Carlo tree search that treats each state of the search tree as a multi-armed bandit. During the first stage of simulation, actions are selected by the UCB1 algorithm (Auer et al., 2002) by the tree policy. Each action value is augmented by an exploration bonus that is highest for rarely visited state-action pairs, and the algorithm selects the action a^* maximising the augmented value,

$$Q^{\oplus}(s,a) = Q(s,a) + c\sqrt{\frac{2\log N(s)}{N(s,a)}} \tag{16}$$

$$a^* = \operatorname*{argmax}_a Q^{\oplus}(s,a) \tag{17}$$

where c is a scalar constant controlling the trade-off between exploration and exploitation.

The performance of UCT can often be significantly improved by incorporating domain knowledge into the default policy (Gelly et al., 2006). The UCT algorithm, using a carefully chosen default policy, has outperformed previous approaches to search in a variety of challenging games, including Go (Gelly et al., 2006), General Game Playing (Finnsson and Björnsson, 2008), Amazons (Lorentz, 2008), Lines of Action (Winands and Björnsson, 2009), multi-player card games (Schäfer, 2008; Sturtevant, 2008), and real-time strategy games (Balla and Fern, 2009).

4.1.2 Beyond Monte-Carlo Tree Search

In Monte-Carlo tree search, states are represented individually. The search tree is based on table lookup, where each node stores the value of exactly one state. Unlike table lookup, only some visited states are stored in the search tree. Once all states have been visited and added into the search tree, Monte-Carlo tree search is equivalent to Monte-Carlo control using table lookup (see Section 2.2), applied to the subproblem starting from s_t; and with the same guarantees of convergence to the optimal policy. Just like table lookup, Monte-Carlo tree search cannot generalise online between related states. Positions in the search tree are evaluated independently: there is no generalisation between similar nodes in the search

Algorithm 2 Linear TD Search

$\theta \leftarrow 0$ ▷ Initialise parameters
procedure SEARCH(s_0)
  while time available **do**
    $e \leftarrow 0$ ▷ Clear eligibility trace
    $s \leftarrow s_0$
    $a \leftarrow \epsilon$–greedy$(s; Q)$
    while s is not terminal **do**
      $s' \sim \mathcal{P}^a_{ss'}$ ▷ Sample state transition
      $r \leftarrow \mathcal{R}^a_{ss'}$ ▷ Sample reward
      $a' \leftarrow \epsilon$–greedy$(s'; Q)$
      $\delta Q \leftarrow r + Q(s', a') - Q(s, a)$ ▷ TD-error
      $\theta \leftarrow \theta + \alpha \delta Q e$ ▷ Update weights
      $e \leftarrow \lambda e + \phi(s, a)$ ▷ Update eligibility trace
      $s \leftarrow s', a \leftarrow a'$
    end while
  end while
  return $\operatorname{argmax}_a Q(s_0, a)$
end procedure

tree; no generalisation to new nodes in the search tree; and no generalisation to positions encountered by the default policy.

Our new method, TD search, uses value function approximation instead of using a search tree. In this approach, the outcome of a single simulation from state s can be used to update the value function for a large number of similar states to s. As a result, TD search can be much more efficient given the same number of simulations. Furthermore, the value function approximates the value of *any* position $s \in \mathcal{S}$, even if it has never been visited. In principle, this means that every step of each simulation can be informed by the value function, without ever exiting the agent's knowledge-base; although in practice, an informed default policy may still be beneficial.

In addition, Monte-Carlo search must wait many time-steps until the final outcome of a simulation is known. This outcome depends on all of the agent's decisions, and on the environment's uncertain responses to those decisions, throughout the simulation. In our framework, we use TD learning instead of Monte-Carlo evaluation, so that the value function can bootstrap from subsequent values. In reinforcement learning, bootstrapping often provides a substantial reduction in variance and an improvement in performance (Singh and Dayan, 1998; Sutton and Barto, 1998). TD search brings this advantage of bootstrapping to simulation-based search.

TD search uses value function approximation and bootstrapping, which both introduce bias into the value function estimate. In particular, an approximate value function cannot usually represent the optimal value function. In principle, this bias could be reduced by using a richer function approximator, for example by including table lookup features in the representation. However, when simulation-based search is applied to large state spaces, the vast majority of states are visited a small number of times, and the variance in their values typically dominates the bias. In these cases the reduced variance of TD search may be much more important than the unbiased evaluation of Monte-Carlo search.

4.2 Linear Temporal-Difference Search

Linear TD search is a simulation-based search algorithm in which the value function is updated online, from simulated experience, by linear TD learning. Each search begins from a root state s_0. The agent simulates many episodes of experience from s_0, by sampling from its current policy $\pi_u(s,a)$, and from a transition model $\mathcal{P}^a_{ss'}$ and reward model $\mathcal{R}^a_{ss'}$, until each episode terminates.

Instead of using a search tree, the agent approximates the value function by using features $\phi(s,a)$ and adjustable weights θ_u, using a linear combination $Q_u(s,a) = \phi(s,a) \cdot \theta_u$. After every step u of simulation, the agent updates the parameters by TD learning, using the TD(λ) algorithm.

The weights are initialised to zero at the first real time-step. At each subsequent real time-step, the weights are reused, so that the value function computed by the last search at time-step $t-1$ provides the initial value function for the new search at time-step t.

The agent selects actions by using an ϵ–greedy policy $\pi_u(s,a)$ that with probability $1-\epsilon$ maximises the current value function $Q_u(s,a)$, and with probability ϵ selects a random action. As in the Sarsa algorithm, this interleaves policy evaluation with policy improvement, with the aim of finding the policy that maximises expected total reward from s_0, given the current model of the environment.

Linear TD search applies the linear Sarsa(λ) algorithm to the subgame or sub-MDP that starts from the state s_0, and thus has the same convergence properties as linear Sarsa(λ), i.e. continued chattering but no divergence (Gordon, 1996) (see Section 2.4). We note that other online, incremental reinforcement learning algorithms could be used in place of Sarsa(λ), for example policy gradient methods (Sutton et al., 2000), if guaranteed convergence were required. However, the computational simplicity of Sarsa is highly desirable during online search.

4.3 Temporal-Difference Search and Monte-Carlo Tree Search

TD search is a general planning method that includes a spectrum of different algorithms. At one end of the spectrum, we can set $\lambda = 1$ to give Monte-Carlo search algorithms, or alternatively we can set $\lambda < 1$ to bootstrap from successive values. We can use table lookup, or we can generalise between states by using abstract features.

In order to reproduce Monte-Carlo tree search, we use $\lambda = 1$ to backup values directly from the final return, without bootstrapping (see Section 2.3). We use one table lookup feature $I^{S,A}$ for each state S and each action A,

$$I^{S,A}(s,a) = \begin{cases} 1 & \text{if } s = S \text{ and } a = A \\ 0 & \text{otherwise} \end{cases} \tag{18}$$

We also use a step-size schedule of $\alpha(s,a) = 1/N(s,a)$, where $N(s,a)$ counts the number of times that action a has been taken from state s. This computes the mean return of all simulations in which action a was taken from state s, in an analogous fashion to Monte-Carlo evaluation. Finally, in order to grow the search tree incrementally, in each simulation we add one new feature $I^{S,A}$ for every action A, for the first visited state S that is not already represented by table lookup features.

In addition, temporal-difference search can incorporate more sophisticated exploration policies than ϵ–greedy. For example, one could construct upper confidence bounds on the

value function, and select actions optimistically with respect to these upper confidence bounds. This could give a more systematic search, analogous to the UCT algorithm.

4.4 Temporal-Difference Search in Computer Go

As we saw in Section 3, local shape features provide a simple but effective representation for some intuitive Go knowledge. The value of each shape can be learnt offline, using TD learning and training by self-play, to provide general knowledge about the game of Go. However, the value function learnt in this way is rather myopic: each square region of the board is evaluated independently, without any knowledge of the global context.

Local shape features can also be used during TD search. Although the features themselves are very simple, TD search is able to learn the value of each feature in the current board context. This can significantly increase the representational power of local shape features: a shape may be bad in general, but good in the current situation. By training from simulated experience, starting from the current state, the agent can focus on what works well *now*.

Local shape features provide a simple but powerful form of generalisation between similar positions. Unlike Monte-Carlo tree search, which evaluates each state independently, the value θ_i of a local shape ϕ_i is reused in a large class of related states $\{s : \phi_i(s) = 1\}$ in which that particular shape occurs. This enables TD search to learn an effective value function from fewer simulations than is possible with Monte-Carlo tree search.

In Section 3 we were able to exploit the symmetries of the Go board by using weight sharing. However, by starting our simulations from the current position, we break these symmetries. The vast majority of Go positions are asymmetric, so that for example the value of playing in the top-left corner will be significantly different to playing in the bottom-right corner. Thus, we do not utilise any form of weight-sharing during TD search. However, local shape features that consist entirely of empty intersections are assumed to be neutral and are removed from the representation.[7]

We apply the TD search algorithm to 9×9 computer Go using 1×1 to 3×3 local shape features. We modify the basic TD search algorithm to exploit the probabilistic nature of the value function, by using logistic NTD2(0) with an ϵ–greedy policy and default parameters of $\alpha = 0.1$ and $\epsilon = 0.1$,

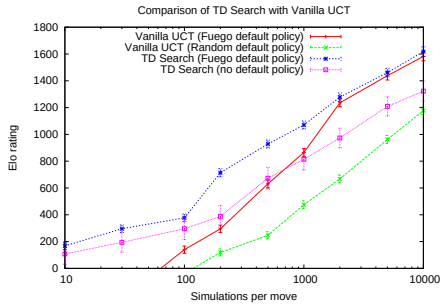
$$V(s) = \sigma(\phi(s) \cdot \theta) \tag{19}$$

$$\Delta\theta = \alpha \frac{\phi(s_t)}{||\phi(s_t)||^2}(V(s_{t+2}) - V(s_t)) \tag{20}$$

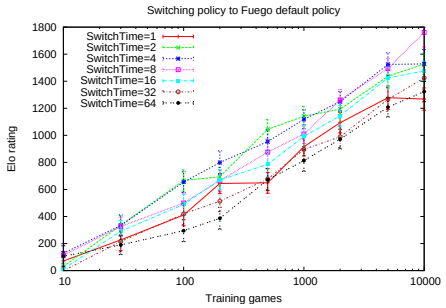
As discussed earlier, one could imagine using a more systematic exploration policy, for example by maximising an upper confidence bound on the value function. However, initial results using this approach were discouraging (Silver, 2009), and therefore we focus on simple ϵ–greedy in the experiments below.

Once each TD search is complete, move a is selected greedily, with no exploration, so as to maximise (Black) or minimise (White) the afterstate value, $V(s \circ a)$.

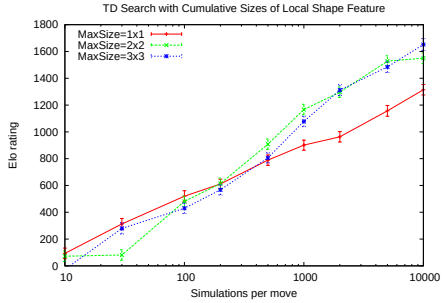
[7] If empty shapes are used, then the algorithm is less effective in opening positions, as the majority of credit is assigned to features corresponding to open space.



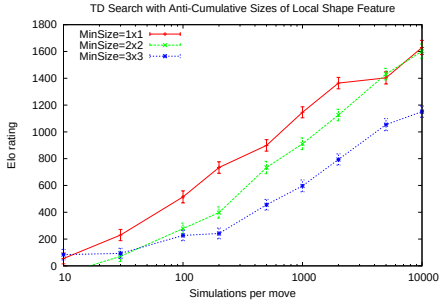
(a) Comparison of TD search and vanilla UCT



(b) Performance of TD search when switching to the *Fuego* default policy. The number of moves at which the switch occurred was varied between 1 and 64.



(c) Cumulative sizes: 1×1; 1×1 and 2×2; and 1×1, 2×2 and 3×3



(d) Anti-cumulative sizes: 1×1, 2×2 and 3×3; 1×1 and 2×2; and 3×3

Fig. 7 TD search in 9×9 Go.

4.5 A Case Study of TD Search in 9×9 Go

We implemented the TD search algorithm in our Go program *RLGO 2.4*. We ran a tournament between different versions of *RLGO*, for a variety of different parameter settings, and a variety of different simulations per move (i.e. varying the search effort). Each Swiss-style tournament[8] consisted of at least 200 games for each version of *RLGO*. After all matches were complete, the results were analysed by the *bayeselo* program (Coulom, 2008) to establish an Elo rating for every program. Two benchmark programs were included in each tournament. First, we included *GnuGo 3.7.10*, set to level 10 (strong, default playing strength), which was assigned an anchor rating of 1800 Elo in all cases. Second, we used an implementation of UCT in *Fuego 0.1* (Müller and Enzenberger, 2009) that we refer to as *vanilla UCT*. This implementation was based on the UCT algorithm, with RAVE and heuristic prior knowledge extensions turned off. Vanilla UCT uses the handcrafted default policy in *Fuego*. The UCT parameters were set to the best reported values for the original *MoGo* program (Gelly et al., 2006): exploration constant $= 1$, first play urgency $= 1$.

4.5.1 Default Policy

The basic TD search algorithm uses no prior knowledge in its simulation policy. One way to incorporate prior knowledge is to switch to a handcrafted default policy, as in the Monte-Carlo tree search algorithm. We ran an experiment to determine the effect on performance of switching to the default policy from *Fuego 0.1* after a constant number of moves T. The results are shown in Figure 7(b).

Switching policy was consistently most beneficial after 2-8 moves, providing around a 300 Elo improvement over no switching. This suggests that the knowledge contained in the local shape features is most effective when applied close to the root, and that the general domain knowledge encoded by the handcrafted default policy is more effective in positions far from the root.

We also compared the performance of TD search against the vanilla UCT implementation in *Fuego 0.1*. We considered two variants of each program, with and without a handcrafted default policy. The same default policy from *Fuego* was used in both programs. When using the default policy, the TD search algorithm switched to the *Fuego* default policy after $T = 6$ moves. When not using the default policy, the ϵ–greedy policy was used throughout all simulations. The results are shown in Figure 7(a).

The basic TD search algorithm, which utilises minimal domain knowledge based only on the grid structure of the board, significantly outperformed vanilla UCT with a random default policy. When using the *Fuego* default policy, TD search again outperformed vanilla UCT, although the difference was not significant beyond 2,000 simulations per move.

In our subsequent experiments, we switched to the *Fuego* default policy after $T = 6$ moves. This had the additional benefit of increasing the speed of our program by an order of magnitude, from around 200 simulations per second to 2,000 simulations per second on a 2.4 GHz processor. For comparison, the vanilla UCT implementation in *Fuego 0.1* executed around 6,000 simulations per second.

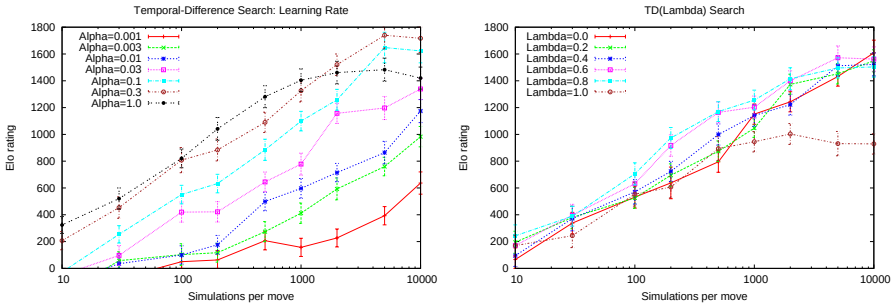
4.5.2 Local Shape Features

The local shape features that we use in our experiments are quite naive: the majority of shapes and tactics described in Go textbooks span considerably larger regions of the board than 3×3 squares. When used in a standard reinforcement learning context, the local shape features achieved a rating of around 1200 Elo (see Section 3). However, when the same representation was used in TD search, combining the 1×1 and 2×2 local shape features achieved a rating of almost 1700 Elo with just 10,000 simulations per move, more than vanilla UCT with an equivalent number of simulations (Figure 7(c)).

The importance of temporality is aptly demonstrated by the 1×1 features. Using TD learning, a static evaluation function based only on these features achieved a rating of just 200 Elo (see Section 3). However, when the feature weights are adapted dynamically, these simple features are often sufficient to identify the critical moves in the current position. TD search increased the performance of the 1×1 features to 1200 Elo, a similar level of performance to TD learning with a million 1×1 to 3×3 features.

Surprisingly, including the more detailed 3×3 features provided no statistically significant improvement. However, we recall from Figure 5(b), when using the standard paradigm of TD learning, that there was an initial period of rapid 2×2 learning, followed by a slower period of 3×3 learning. Furthermore we recall that, without weight sharing, this transition

[8] Matches were randomly selected with a bias towards programs with a similar number of wins.



(a) Varying the step-size α.

(b) TD search using logistic NTD2(λ) and varying λ.

Fig. 8 Parameter study for TD search in 9×9 Go.

took place after many thousands of simulations. This suggests that our TD search results correspond to the steep region of the learning curve, and that the rate of improvement is likely to flatten out with additional simulations.

4.5.3 Parameter Study

In our next experiment we varied the step-size parameter α (Figure 8(a)). The results clearly show that an aggressive learning rate is most effective across a wide range of simulations per move. However, the rating improvement for the most aggressive learning rates flattened out with additional computation, after 1,000 simulations per move for $\alpha = 1$, and after 5,000 simulations per move for $\alpha = 0.1$ and $\alpha = 0.3$. This suggests that a decaying step-size schedule might achieve better performance with longer search times.

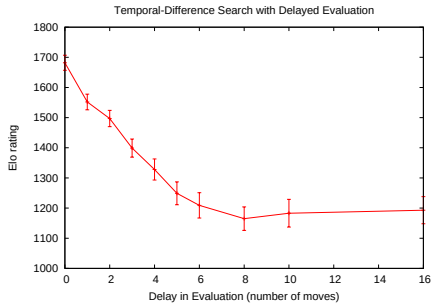
We extend the TD search algorithm to utilise TD(λ), using the eligibility trace update equation described in Section 3. We study the effect of the temporal-difference parameter λ in Figure 8(b). Bootstrapping ($\lambda < 1$) provided a significant performance benefit.

Previous work in simulation-based search has largely been restricted to Monte-Carlo methods (Tesauro and Galperin, 1996; Kocsis and Szepesvari, 2006; Gelly et al., 2006; Gelly and Silver, 2007; Coulom, 2007). However, our results suggest that generalising these approaches to temporal-difference methods may provide significant benefits when value function approximation is used.

4.5.4 Temporality and Temporal Coherence

Successive positions are strongly correlated in the game of Go. Each position changes incrementally, by just one new stone at every non-capturing move. Groups and fights develop, providing specific shapes and tactics that may persist for a significant proportion of the game, but are unique to this game and are unlikely to ever be repeated in this combination. We conducted two experiments to disrupt this temporal coherence, so as to gain some insight into its effect on TD search.

In our first experiment, we selected moves according to an old value function from a previous search. At move number t, the agent selects the move that maximises the value function that it computed at move number $t - k$, for some move gap $0 \leq k < t$. The results, shown in Figure 9(a), indicate the rate at which the global context changes. The value



(a) TD search with 10,000 simulations/move. The results of the latest search are only used after a k move delay.

(b) Comparison of TD search when the weights are reset to zero at the start of each search, to when the weights are reused from the previous search.

Fig. 9 Studying the effect of temporality in TD search.

function computed by the search is highly specialised to the current situation. When it was applied to the position that arose just 6 moves later, the performance of *RLGO*, using 10,000 simulations per move, dropped from 1700 to 1200 Elo, the same level of performance that was achieved by standard TD learning (see Section 3). This also explains why it is beneficial to switch to a handcrafted default policy after around 6 moves (see Figure 7(b)).

In our second experiment, instead of reusing the weights from the last search, we reset the weights θ to zero at the beginning of every search, so as to disrupt any transfer of knowledge between successive moves. The results are shown in Figure 9(b). Resetting the weights dramatically reduced the performance of our program by between 400–800 Elo. This suggests that a very important aspect of TD search is its ability to accumulate knowledge over several successive, highly related positions.

4.5.5 Board Sizes

Finally, we compared the performance of TD search with vanilla UCT, on board sizes from 5×5 up to 15×15. As before, the same default policy was used in both cases, beyond the search tree for vanilla UCT, and after $T = 6$ moves for TD search. The results are shown in Figure 10.

In 5×5 Go, vanilla UCT was able to play near-perfect Go, and significantly outperformed the approximate evaluation used by TD search. In 7×7 Go, the results were inconclusive, with both programs performing similarly with 10,000 simulations per move. However, on larger board sizes, TD search outperformed vanilla UCT by a margin that increased with larger board sizes. In 15×15 Go, using 10,000 simulations per move, TD search outperformed vanilla UCT by around 500 Elo. This suggests that the importance of generalising between states increases with larger branching factors and larger search spaces. Taking account of the 3 to 4-times slower execution speed of TD search compared to vanilla UCT (Figure 11), the two algorithms perform comparably on larger board sizes.

These experiments used the same default parameters from the 9×9 experiments. It is likely that both TD search and vanilla UCT could be improved by retuning the parameters to the different board sizes. Note also that the strength of the reference program, GnuGo, varies considerably across board sizes.

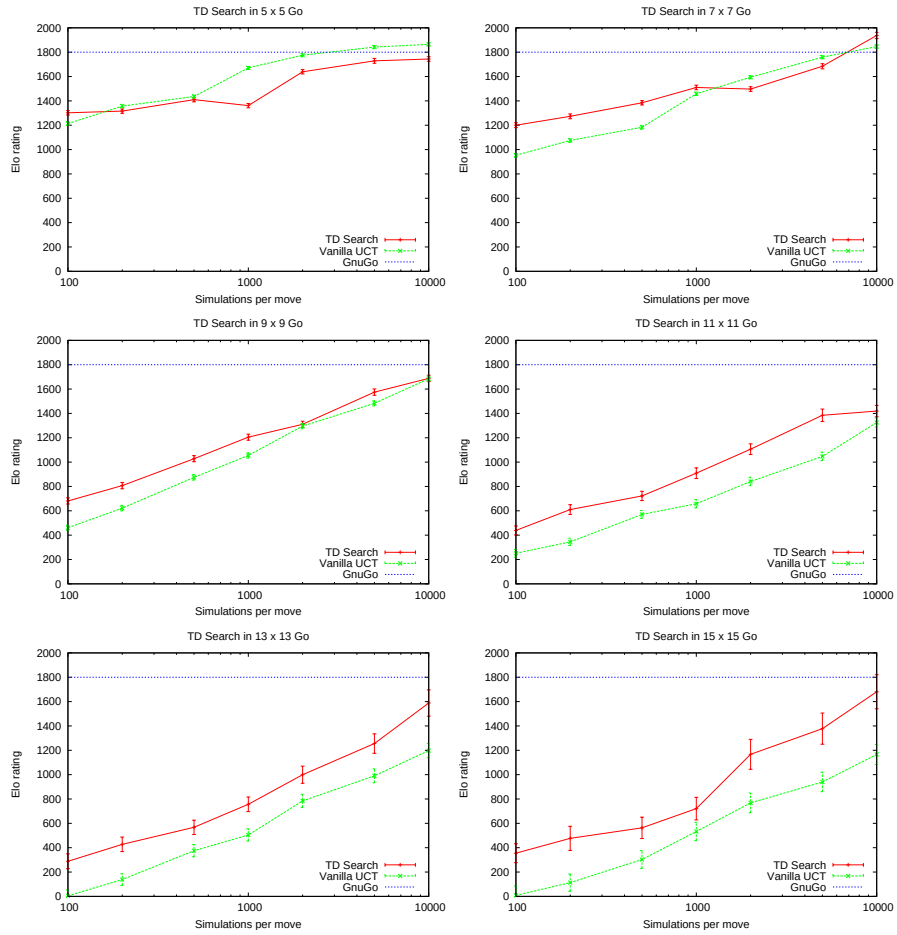


Fig. 10 Comparison of TD search and vanilla UCT with different board sizes.

5 Dyna-2: Integrating Long and Short-Term Memories

In many problems, *learning* and *search* must be combined together in order to achieve good performance. Learning algorithms slowly extract knowledge, from the complete history of training data, that applies very generally throughout the domain. Search algorithms use and extend this knowledge, rapidly and online, so as to evaluate local states more accurately. Learning and search often interact in a complex and surprising fashion, and the most successful approaches integrate both processes together (Schaeffer, 2000; Fürnkranz, 2001).

In Section 3 we used TD learning to extract general domain knowledge from games of self-play (Schraudolph et al., 1994; Enzenberger, 1996; Dahl, 1999; Enzenberger, 2003; Silver et al., 2007). In Section 4, we used TD search to form a local value function approximation, without any general domain knowledge (see Section 4). In this section we develop a unified architecture, *Dyna-2*, that combines both TD learning and TD search.

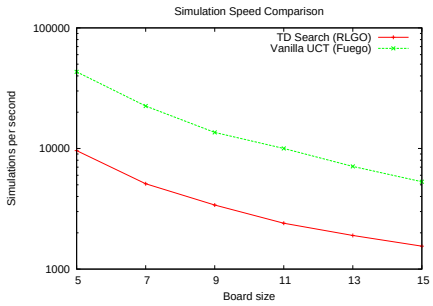


Fig. 11 Speed comparison between TD search and vanilla UCT with different board sizes.

5.1 Dyna and Dyna-2

Sutton's *Dyna* architecture (Sutton, 1990) applies TD learning both to real experience and to simulated experience. The agent learns a model of the MDP from real experience, and updates its action value function from both real and simulated experience. For example, the *Dyna-Q* algorithm remembers all state transitions and rewards from all visited states and actions in its real experience. Before selecting the next real action, the agent executes some number of planning steps. At each step, a previously visited state and action is selected, and a state transition and reward are sampled from the memorised experience. The action value is updated by applying TD(λ) to each sampled transition (planning), and also to each real transition (learning).

The key new idea of Dyna-2 is to maintain two separate memories: a long-term memory that is learnt from real experience; and a short-term memory that is used during search, and is updated from simulated experience. Furthermore, we specifically consider simulations which are drawn sequentially in episodes that start from the current state. As we saw in the previous section, this helps focus learning on the future distribution from time-step t onwards, rather than the global distribution over all possible states.

5.2 Long and Short-Term Memories

Domain knowledge contains many general rules, but even more special cases. Chess grandmaster Kevin Spraggett once said, "I spent the first half of my career learning the principles for playing strong chess and the second half learning when to violate them" (Schaeffer, 1997). Long and short-term memories can be used to represent both aspects of knowledge.

We define a *memory* $M = (\phi, \theta)$ to be a vector of features ϕ, and a vector of corresponding parameters θ. The feature vector $\phi(s, a)$ compactly represents the state s and action a, and provides an abstraction of the state and action space. The parameter vector θ is used to approximate the value function, by forming a linear combination $\phi(s, a) \cdot \theta$ of the features and parameters in M.

In our architecture, the agent maintains two distinct memories: a *long-term memory* $M = (\phi, \theta)$ and a *short-term memory* $\overline{M} = (\overline{\phi}, \overline{\theta})$.[9] The agent also maintains two distinct

[9] These names are suggestive of each memory's function, but are not related to biological long and short-term memory systems. There is also no relationship to the Long Short-Term Memory algorithm for training recurrent neural networks (Hochreiter and Schmidhuber, 1997).

approximations to the value function. The *long-term value function*, $Q(s,a)$, uses only the long-term memory to approximate the true value function $Q^\pi(s,a)$. The *short-term value function*, $\overline{Q}(s,a)$, uses *both* memories to approximate the true value function, by forming a linear combination of both feature vectors with both parameter vectors,

$$Q(s,a) = \phi(s,a) \cdot \theta \tag{21}$$

$$\overline{Q}(s,a) = \phi(s,a) \cdot \theta + \overline{\phi}(s,a) \cdot \overline{\theta} \tag{22}$$

The long-term memory is used to represent *general* knowledge about the problem, i.e. knowledge that is independent of the agent's current state. For example, in chess the long-term memory could know that a bishop is worth 3.5 pawns. The short-term memory is used to represent *local* knowledge about the problem, i.e. knowledge that is specific to the agent's current region of the state space. The short-term memory is used to *correct* the long-term value function, representing adjustments that provide a more accurate local approximation to the true value function. For example, in a closed endgame position, the short-term memory could know that the black bishop is worth 1 pawn less than usual. These corrections may actually hurt the global approximation to the value function, but if the agent continually adjusts its short-term memory to match its current state, then the overall quality of approximation can be significantly improved.

5.3 Dyna-2

The core idea of Dyna-2 is to combine TD learning with TD search, using long and short-term memories. The long-term memory is updated from real experience, and the short-term memory is updated from simulated experience, in both cases using the TD(λ) algorithm. We denote short-term parameters with a bar, $\overline{x}$, and long-term parameters with no bar, x.

At the beginning of each real episode, the contents of the short-term memory are cleared, $\overline{\theta} = 0$. At each real time-step t, before selecting its action a_t, the agent executes a simulation-based search. Many simulations are launched, each starting from the agent's current state s_t. After each step of computation u, the agent updates the weights of its short-term memory from its simulated experience $(s_u, a_u, r_{u+1}, s_{u+1}, a_{u+1})$, using the TD($\lambda$) algorithm. The TD-error is computed from the short-term value function, $\overline{\delta}_u = r_{u+1} + \overline{Q}(s_{u+1}, a_{u+1}) - \overline{Q}(s_u, a_u)$. Actions are selected using an $\overline{\epsilon}$–greedy policy that maximises the short-term value function $a_u = \operatorname{argmax}_b \overline{Q}(s_u, b)$. This search procedure continues for as much computation time as is available.

When the search is complete, the short-term value function represents the agent's best local approximation to the optimal value function. The agent then selects a real action a_t using an ϵ–greedy policy that maximises the short-term value function $a_t = \operatorname{argmax}_b \overline{Q}(s_t, b)$. After each time-step, the agent updates its long-term value function from its real experience $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$, again using the TD($\lambda$) algorithm. This time, the TD-error is computed from the long-term value function, $\delta_t = r_{t+1} + Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$. In addition, the agent uses its real experience to update its state transition model $\mathcal{P}^a_{ss'}$ and its reward model $\mathcal{R}^a_{ss'}$. The complete algorithm is described in pseudocode in Algorithm 3.

The Dyna-2 architecture learns from both the past and the future. The long-term memory is updated from the agent's actual past experience. The short-term memory is updated from sample episodes of what could happen in the future. Combining both memories together provides a much richer representation than is possible with a single memory.

Algorithm 3 Episodic Dyna-2

procedure LEARN
  Initialise $\mathcal{P}^a_{ss'}, \mathcal{R}^a_{ss'}$ ▷ State transition and reward models
  $\theta \leftarrow 0$ ▷ Clear long-term memory
  loop
    $s \leftarrow s_0$ ▷ Start new episode
    $\overline{\theta} \leftarrow 0$ ▷ Clear short-term memory
    $e \leftarrow 0$ ▷ Clear eligibility trace
    SEARCH(s)
    $a \leftarrow \epsilon$–greedy$(s; \overline{Q})$
    while s is not terminal **do**
      Execute a, observe reward r, state s'
      $\mathcal{P}^a_{ss'}, \mathcal{R}^a_{ss'} \leftarrow$ UPDATEMODEL(s, a, r, s')
      SEARCH(s')
      $a' \leftarrow \epsilon$–greedy$(s'; \overline{Q})$
      $\delta Q \leftarrow r + Q(s', a') - Q(s, a)$ ▷ TD-error
      $\theta \leftarrow \theta + \alpha \delta Q e$ ▷ Update weights
      $e \leftarrow \lambda e + \phi$ ▷ Update eligibility trace
      $s \leftarrow s', a \leftarrow a'$
    end while
  end loop
end procedure

procedure SEARCH(s)
  while time available **do**
    $\overline{e} \leftarrow 0$ ▷ Clear eligibility trace
    $a \leftarrow \overline{\epsilon}$–greedy$(s; \overline{Q})$
    while s is not terminal **do**
      $s' \sim \mathcal{P}^a_{ss'}$ ▷ Sample state transition
      $r \leftarrow \mathcal{R}^a_{ss'}$ ▷ Sample reward
      $a' \leftarrow \overline{\epsilon}$–greedy$(s'; \overline{Q})$
      $\overline{\delta} Q \leftarrow r + \overline{Q}(s', a') - \overline{Q}(s, a)$ ▷ TD-error
      $\overline{\theta} \leftarrow \overline{\theta} + \overline{\alpha} \overline{\delta} Q \overline{e}$ ▷ Update weights
      $\overline{e} \leftarrow \overline{\lambda} \overline{e} + \overline{\phi}$ ▷ Update eligibility trace
      $s \leftarrow s', a \leftarrow a'$
    end while
  end while
end procedure

A particular instance of Dyna-2 must specify learning parameters: a set of features ϕ for the long-term memory; a temporal-difference parameter λ; an exploration rate ϵ and a learning rate α. Similarly, it must specify the equivalent search parameters: a set of features $\overline{\phi}$ for the short-term memory; a temporal-difference parameter $\overline{\lambda}$; an exploration rate $\overline{\epsilon}$ and a learning rate $\overline{\alpha}$.

The Dyna-2 architecture subsumes a large family of learning and search algorithms. If there is no short-term memory, $\overline{\phi} = \emptyset$, then the search procedure has no effect and may be skipped. This results in the linear Sarsa algorithm (see Section 2.4). If there is no long-term memory, $\phi = \emptyset$, then Dyna-2 reduces to the TD search algorithm. As we saw in Section 4, this algorithm itself subsumes a variety of simulation-based search algorithms such as Monte-Carlo tree search.

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 agent's long-term memory is

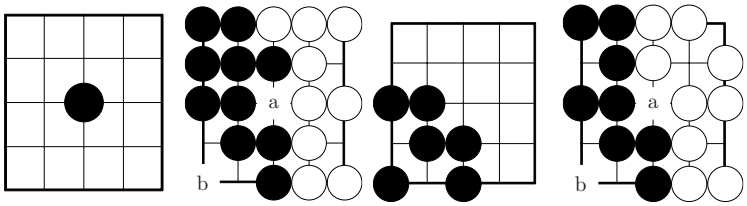


Fig. 12 a) A 1×1 local shape feature with a central black stone. This feature acquires a strong positive value in the long-term memory. b) In this position, move b is the winning move. Using only 1×1 local shape features, the long-term memory suggests that move a should be played. The short-term memory will quickly learn to correct this misevaluation, reducing the value of a and increasing the value of b. c) A 3×3 local shape feature making two eyes in the corner. This feature acquires a positive value in the long-term memory. d) Black to play, using Chinese rules, move a is now the winning move. Using 3×3 features, the long-term memory suggests move b, believing this to be a good shape in general. However, the short-term memory quickly realises that move b is redundant in this context (black already has two eyes) and learns to play the winning move at a.

learnt offline during a preliminary training phase. When the agent is placed into the real environment, it uses its short-term memory to adjust to the current state. Even if the agent's model is inaccurate, each simulation begins from its true current state, which means that the simulations are usually fairly accurate for at least the first few steps. This allows the agent to dynamically correct at least some of the misconceptions in the long-term memory.

5.4 Dyna-2 in Computer Go

We have already seen that local shape features can be used with TD learning, to learn general Go knowledge (see Section 3). We have also seen that local shape features can be used with TD search, to learn the value of shapes in the current situation (see Section 4). The Dyna-2 architecture lets us combine the advantages of both approaches, by using local shape features in both the long and short-term memories.

Figure 12 gives a very simple illustration of long and short-term memories in 5×5 Go. It is usually bad for Black to play on the corner intersection, and so long-term memory learns a negative weight for this feature. However, Figure 12 shows a position in which the corner intersection is the most important point on the board for Black: it makes two eyes and allows the Black stones to live. By learning about the particular distribution of states arising from this position, the short-term memory learns a large positive weight for the corner feature, correcting the long-term memory.

In general, it may be desirable for the long and short-term memories to utilise different features, which are best suited to representing either general or local knowledge. In our computer Go experiments, we focus our attention on the simpler case where both vectors of features are identical, $\phi = \overline{\phi}$. In this special case, the Dyna-2 algorithm can be implemented more efficiently, using just one memory during search. At the start of each real game, the contents of the short-term memory are initialised to the contents of the long-term memory, $\overline{\theta} \leftarrow \theta$, and the short-term value function is computed simply by $\overline{Q}(s,a) = \overline{\phi}(s,a) \cdot \overline{\theta}$. As a result, only one memory and one value function are required during actual play, and the resulting search algorithm (after initialising the short-term memory) reduces to TD search (Algorithm 2).

We compared our algorithm to the vanilla UCT implementation from the *Fuego 0.1* program (Müller and Enzenberger, 2009), as described in Section 4.5. Both *RLGO* and vanilla

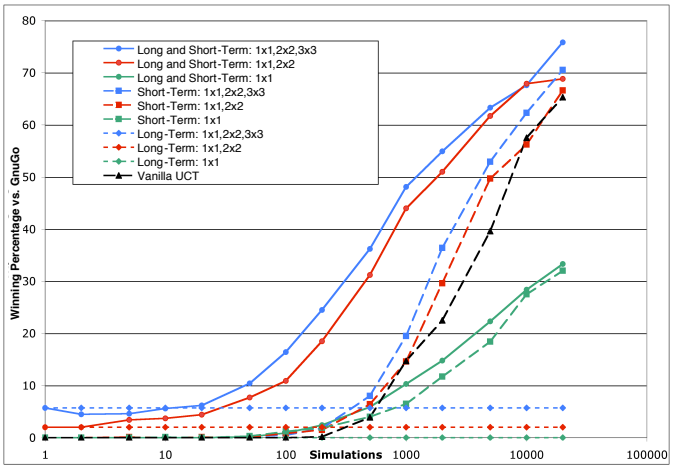


Fig. 13 Winning percentage of *RLGO 2.4* against *GnuGo 3.7.10* (level 0) in 9×9 Go, for different numbers of simulations per move. Local shape features were used in either the long-term memory (dotted lines), the short-term memory (dashed lines), or both memories (solid lines). The long-term memory was trained in a separate offline phase 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 1,000 games.

UCT used an identical default policy. We separately evaluated both *RLGO* and vanilla UCT by running 1,000 game matches against *GnuGo 3.7.10* (level 0).[10]

We compared the performance of several different variants of our algorithm. First, we evaluated the performance of the long-term memory by itself, $\overline{\phi} = \emptyset$, which is equivalent to the TD learning algorithm developed in Section 3. Second, we evaluated the performance of the short-term memory by itself, $\phi = \emptyset$, which is equivalent to the TD search algorithm developed in Section 4. Finally, we evaluated the performance of both long and short-term memories, making use of the full Dyna-2 algorithm. In each case we compared the performance of local shape features of different sizes (see Figure 13).

Using only the long-term memory, *RLGO 2.4* achieved a winning rate of just 5% against *GnuGo*. Using only the short-term memory, *RLGO* achieved better performance per simulation than vanilla UCT, by a small margin, for up to 20,000 simulations per move. *RLGO* outperformed *GnuGo* with 5,000 or more simulations. Using both memories, *RLGO* achieved significantly better performance per move than vanilla UCT, by a wide margin for few simulations per move and by a smaller but significant margin for 20,000 simulations per move. Using both memories, it outperformed *GnuGo* with just 2,000 or more simulations.

Search algorithm	Memory	Elo rating on CGOS
Alpha-beta	Long-term	1350
Dyna-2	Long and short-term	2030
Dyna-2 + alpha-beta	Long and short-term	2130

Table 4 The Elo ratings established by *RLGO 2.4* on the Computer Go Server.

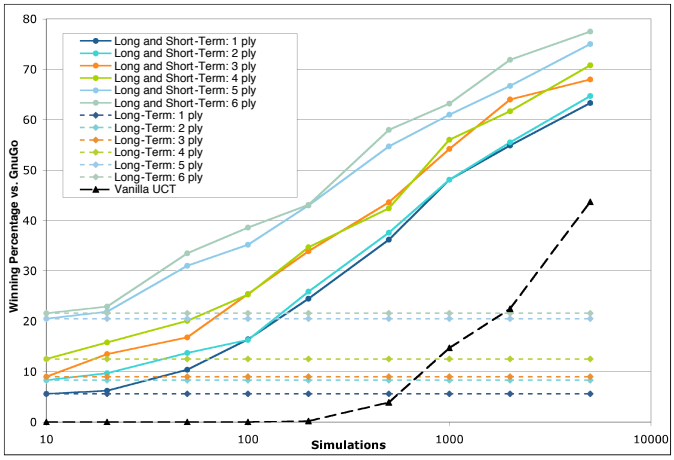


Fig. 14 Winning percentage of *RLGO 2.4* against *GnuGo 3.7.10* (level 0) in 9×9 Go, using a two-phase search with Dyna-2 and alpha-beta. A full-width α-β search was used for move selection, using a value function based on either the long-term memory alone (dotted lines), or both long and short-term memories (solid lines). Using only the long-term memory corresponds to a traditional alpha-beta search. Using both memories, but only a 1-ply search, corresponds to the Dyna-2 algorithm. The long-term memory was trained offline from 100,000 games of self-play. Each point represents the winning percentage over 1,000 games.

6 Two-Phase Search

6.1 Two-Phase Alpha-Beta Search

In games such as chess, checkers and Othello, human world-champion level play has been exceeded by combining a heuristic evaluation function with alpha-beta search. The heuristic function is a linear combination of binary features, and can be learnt offline by TD learning and self-play (Baxter et al., 2000; Schaeffer et al., 2001; Buro, 1999). In Section 3, we saw how this approach could be applied to Go, by using local shape features. In the previous sections we developed a significantly more accurate approximation of the value function, by combining long and short-term memories, using both TD learning and TD search. Can this more accurate value function be successfully used to evaluate positions in a traditional alpha-beta search?

We refer to this approach, in which a simulation-based search is followed by a second complementary search, as a *two-phase search*. We now consider a two-phase search in which

[10] *GnuGo* plays significantly faster at level 0 than at its default level 10, so that results can be collected from many more games. Level 0 is approximately 150 Elo weaker than level 10.

an alpha-beta search is performed after each TD search. As in Dyna-2, after the simulation-based search is complete, the agent selects a real move to play. However, instead of directly maximising the short-term value function, an alpha-beta search is used to find the best move in the depth d minimax tree, where the leaves of the tree are evaluated according to the short-term value function $\overline{Q}(s, a)$.

The two-phase algorithm can also be viewed as an extension to alpha-beta search, in which the evaluation function is dynamically updated. At the beginning of the game, the evaluation function is set to the contents of the long-term memory. Before each alpha-beta search, the evaluation function is re-trained by a TD search. The alpha-beta search then proceeds as usual, but using the updated evaluation function.

We compared the performance of the two-phase search algorithm to a more traditional search algorithm. In the traditional search, the long-term memory $Q(s, a)$ is used as a heuristic function to evaluate leaf positions, as in Section 3. The results are shown in Figure 14. To compare the relative performance relative to computation time, note that a 5-ply alpha-beta search took approximately the same computation time as 1,000 simulations in *RLGO* or 3,000 simulations in *Fuego*.

Dyna-2 outperformed traditional search by a wide margin. Using only 200 simulations per move, *RLGO* exceeded the performance of a full-width 6-ply search. When combined with alpha-beta in the two-phase search algorithm, the results were even better. Alpha-beta provided a substantial performance boost of around 15-20% against GnuGo, which remained approximately constant throughout the tested range of simulations per move. With 5,000 simulations per move, the two-phase algorithm achieved a winning rate of almost 80% against GnuGo. These results suggest that the benefits of alpha-beta search are largely complementary to the simulation-based search.

Finally, we implemented a high-performance version of our two-phase search algorithm in *RLGO 2.4*. In this tournament version, time was dynamically allocated, approximately evenly between the two search algorithms, using an exponentially decaying time control. We extended the TD search to use multiple processors, by sharing the long and short-term memories between processes, and to use pondering, by simulating additional games of self-play during the opponent's thinking time. We extended the alpha-beta search to use several well-known extensions: iterative deepening, transposition table, killer move heuristic, and null-move pruning (Schaeffer, 2000). RLGO competed on the 9×9 Computer Go Server (CGOS), which uses 5 minute time controls, for several hundred games in total. The ratings established by RLGO are shown in Table 4.

Using only an alpha-beta search, based on the long-term memory alone, RLGO established a rating of 1350 Elo. Using Dyna-2, using both long and short-term memories, but no alpha-beta search, RLGO established a rating of 2030 Elo. Using the two-phase search algorithm, including Dyna-2 and also an alpha-beta search, RLGO established a rating of 2130 Elo. If we view the two-phase search as an extension to alpha-beta, then we see that dynamically updating the evaluation function offers dramatic benefits, improving the performance of RLGO by 800 Elo. If we view the two-phase search as an extension to Dyna-2, then the performance improves by a more modest, but still significant 100 Elo.

For comparison, the strongest traditional Go programs (for example *GnuGo* and *Many Faces of Go 11*, based on alpha-beta search and handcrafted pattern databases, have been rated at ~1800 Elo on CGOS. Similarly, the strongest reinforcement learning Go programs (for example *NeuroGo* (Enzenberger, 2003) and *Honte* (Dahl, 1999)), have been rated at ~1850 Elo; and the strongest supervised learning Go programs (for example *Magog* (van der Werf et al., 2002)) have been rated at ~1700 Elo. These previous programs incorporated a great deal of sophisticated handcrafted knowledge about the game of Go, whereas the

Initial value function	Wins vs. *GnuGo*	Error
Zero value, zero count	61%	$\pm$ 2%
Handcrafted prior knowledge	66%	$\pm$ 2%
Long-term value $Q(s,a)$	68%	$\pm$ 2%
Short-term value $\overline{Q}(s,a)$	67%	$\pm$ 3%

Table 5 Two-phase search in *MoGo* using an enhanced Monte-Carlo tree search and initialising the value of new nodes to a given value function. The long-term value function was learnt from 100,000 games; the short-term value function was learnt online from 3,000 simulations per move. *MoGo* was then given an additional 3,000 simulations per move.

domain knowledge in *RLGO* is minimal. *RLGO*'s performance on CGOS is comparable to or exceeds the performance of many unenhanced Monte-Carlo tree search programs. However, it is significantly weaker than the strongest Monte-Carlo tree search programs (for example *Zen*, *Fuego* and *MoGo*), which exploit domain specific knowledge to achieve ratings in excess of 2600 Elo on CGOS (Chaslot et al., 2008).

6.2 Two-Phase Monte-Carlo Tree Search

Finally, we consider a two-phase search in which a TD search is followed by a Monte-Carlo tree search, rather than an alpha-beta search. In this approach, the value of new nodes in the Monte-Carlo tree search is initialised to the short-term value function $\overline{Q}(s,a)$. The count of new nodes is initialised to a constant value M corresponding to the level of confidence in the value function (equivalent to a beta prior). We hooked RLGO's value function into one of the strongest Monte-Carlo tree search programs, *MoGo* (Gelly et al., 2006), enhanced by the RAVE algorithm (Gelly and Silver, 2007), and evaluated its performance in 9×9 Go against *GnuGo* (level 10) when given just 3000 simulations per move, for the best value of M. For comparison, we also evaluated *MoGo* using several other choices of node initialisation: using *MoGo's* handcrafted prior knowledge to initialise nodes; using RLGO's long-term value function $Q(s,a)$ to initialise nodes; and intialising nodes to zero value with zero count. The results are summarised in Table 5; more details can be found in (Gelly and Silver, 2007, 2011; Silver, 2009).

Using the long-term value function produced a significant improvement over the zero value function, and comparable performance to the handcrafted prior knowledge in *MoGo*.[11] Surprisingly, using the short-term value function did not produce any additional performance benefit. This is almost certainly due to the temporality of the short-term memory. In Section 4 we saw that the short-term value function learnt by TD search is specialised to positions occurring up to 6 moves in the future; beyond this point it is no better than a static evaluation function learnt by TD learning. However, high performance Monte-Carlo search programs such as *MoGo* routinely search to depths of 10 or more moves, so that the majority of leaf evaluations are not improved by TD search.

An initial phase of TD learning can also be used to learn a default policy for Monte-Carlo search. The default policy can either act greedily with respect to the long-term value function (Gelly and Silver, 2007), or the policy parameters can be optimised directly (Silver and Tesauro, 2009); the latter approach leading to better results in practice (Huang et al., 2009).

[11] Subsequent versions of *MoGo* improved the handcrafted prior knowledge and produced significantly higher performance.

7 Conclusion

Reinforcement learning is often considered a slow procedure. Outstanding examples of success have, in the past, learnt a value function from months of offline computation. However, this does not need to be the case. Many reinforcement learning methods, such as Monte-Carlo learning and TD learning, are fast, incremental, and scalable. When such a reinforcement learning algorithm is applied to experience simulated from the current state, it produces a high performance search algorithm.

Monte-Carlo tree search algorithms, such as UCT, have recently received much attention. However, this is just one example of a simulation-based search algorithm. There is a spectrum of algorithms that vary from table lookup to highly abstracted state representations, and from Monte-Carlo evaluation to TD learning. Value function approximation can provide rapid generalisation in large problems, and bootstrapping can be advantageous in the presence of function approximation. By varying these dimensions in the TD search algorithm, we have achieved better search efficiency, in 9×9 Go, than a vanilla UCT search. Furthermore, the advantage of TD search increased with larger board sizes.

The Monte-Carlo tree search algorithm retains several advantages over TD search. It can typically execute more simulations per second, and given unlimited time and memory algorithms it will converge on the optimal policy. In the last few years it has been enhanced by domain knowledge and various heuristics and extensions, to produce the first master-level Go programs. In many ways our initial implementation of TD search is more naive: it uses a fixed set of straightforward features, a simplistic ϵ–greedy exploration strategy, a non-adaptive step-size, and a constant policy switching time. The promising results of this basic strategy suggest that the full spectrum of simulation-based methods, not just Monte-Carlo and table lookup, merit further investigation.

References

Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time analysis of the multi-armed bandit problem. *Machine Learning*, 47(2–3):235–256.

Balla, R. and Fern, A. (2009). UCT for tactical assault planning in real-time strategy games. In *21st International Joint Conference on Artificial Intelligence*.

Baxter, J., Tridgell, A., and Weaver, L. (2000). Learning to play chess using temporal differences. *Machine Learning*, 40(3):243–263.

Buro, M. (1999). From simple features to sophisticated evaluation functions. In *1st International Conference on Computers and Games*, pages 126–145.

Chaslot, G., Chatriot, L., Fiter, C., Gelly, S., Hoock, J., Perez, J., Rimmel, A., and Teytaud, O. (2008). Combining expert, online, transient and online knowledge in Monte-Carlo exploration. In *8th European Workshop on Reinforcement Learning*.

Coulom, R. (2006). Efficient selectivity and backup operators in Monte-Carlo tree search. In *5th International Conference on Computer and Games*, pages 72–83.

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

Coulom, R. (2008). Whole-history rating: A bayesian rating system for players of time-varying strength. In *Computers and Games*, pages 113–124.

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

Dayan, P. and Sejnowski, T. (1994). TD(λ) converges with probability 1. *Machine Learning*, 14(1):295–301.

Elo, A. (1978). *The Rating Of Chess Players, Past & Present*. Arco.

Enzenberger, M. (1996). The integration of a priori knowledge into a Go playing neural network. *http://www.cs.ualberta.ca/ emarkus/neurogo/neurogo1996.html*.

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

Finnsson, H. and Björnsson, Y. (2008). Simulation-based approach to general game playing. In *23rd Conference on Artificial Intelligence*, pages 259–264.

Fürnkranz, J. (2001). Machine learning in games: A survey. In *Machines That Learn to Play Games*, pages 11–59. Nova Science Publishers.

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

Gelly, S. and Silver, D. (2011). Monte-Carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175:1856–1875.

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

Gordon, G. (1996). Chattering in SARSA(lambda) - a CMU learning lab internal report. Technical report, Carnegie Mellon University.

Haykin, S. (1996). *Adaptive Filter Theory*. Prentice Hall.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

Huang, S., Coulom, R., and Lin, S. (2009). Monte-Carlo simulation balancing in practice. In *7th International Conference on Computers and Games*, pages 119–126.

Jordan, M. (1995). Why the logistic function? A tutorial discussion on probabilities and neural networks. Technical Report Computational Cognitive Science Report 9503, Massachusetts Institute of Technology.

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

Littman, M. (1994). Markov games as a framework for multi-agent reinforcement learning. In *11th International Conference on Machine Learning*, pages 157–163.

Littman, M. (1996). *Algorithms for Sequential Decision Making*. PhD thesis, Brown University.

Lorentz, R. (2008). Amazons discover monte-carlo. In *Computers and Games*, pages 13–24.

Matthews, C. (2003). *Shape up*. GoBase.

Mayer, H. (2007). Board representations for neural Go players learning by temporal difference. In *IEEE Symposium on Computational Intelligence and Games*.

Müller, M. (2002). Computer Go. *Artificial Intelligence*, 134:145–179.

Müller, M. and Enzenberger, M. (2009). Fuego – an open-source framework for board games and Go engine based on Monte-Carlo tree search. Technical Report TR09-08, University of Alberta, Department of Computing Science.

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

Schaeffer, J. (1997). *One Jump Ahead: Challenging Human Supremacy in Checkers*. Springer Verlag.

Schaeffer, J. (2000). The games computers (and people) play. *Advances in Computers*, 50:189–266.

Schaeffer, J., Hlynka, M., and Jussila, V. (2001). Temporal difference learning applied to a high-performance game-playing program. In *17th International Joint Conference on Artificial Intelligence*, pages 529–534.

Schäfer, J. (2008). The UCT algorithm applied to games with imperfect information. Diploma Thesis. Otto-von-Guericke-Universität Magdeburg.

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

Silver, D. (2009). *Reinforcement Learning and Simulation Based Search in the Game of Go*. PhD thesis, University of Alberta.

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

Silver, D. and Tesauro, G. (2009). Monte-Carlo simulation balancing. In *26th International Conference on Machine Learning*, pages 119–126.

Singh, S. and Dayan, P. (1998). Analytical mean squared error curves for temporal difference learning. *Machine Learning*, 32(1):5–40.

Singh, S., Jaakkola, T., Littman, M., and Szepesvari, C. (2000). Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning*, 38:287–308.

Stern, D., Herbrich, R., and Graepel, T. (2006). Bayesian pattern ranking for move prediction in the game of Go. In *23rd International Conference of Machine Learning*, pages 873–880.

Stoutamire, D. (1991). *Machine Learning, Game Play, and Go*. PhD thesis, Case Western Reserve University.

Sturtevant, N. (2008). An analysis of UCT in multi-player games. In *6th International Conference on Computers and Games*, pages 37–49.

Sutton, R. (1984). *Temporal credit assignment in reinforcement learning*. PhD thesis, University of Massachusetts.

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

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

Sutton, R. and Barto, A. (1998). *Reinforcement Learning: an Introduction*. MIT Press.

Sutton, R., McAllester, D., Singh, S., and Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *In Advances in Neural Information Processing Systems 12*, pages 1057–1063. MIT Press.

Tesauro, G. (1994). TD-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215–219.

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

Tsitsiklis, J. (2002). On the convergence of optimistic policy iteration. *Journal of Machine Learning Research*, 3:59–72.

van der Werf, E., Uiterwijk, J., Postma, E., and van den Herik, J. (2002). Local move prediction in Go. In *3rd International Conference on Computers and Games*, pages 393–412.

Veness, J., Silver, D., Blair, A., and Uther, W. (2009). Bootstrapping from game tree search. In *Advances in Neural Information Processing Systems 19*.

Widrow, B. and Stearns, S. (1985). *Adaptive Signal Processing*. Prentice-Hall.

Winands, M. and Björnsson, Y. (2009). Evaluation function based Monte-Carlo LOA. In *12th Advances in Computer Games Conference*.