Combining TD-learning with Cascade-correlation Networks

François Rivest FRANCOIS.RIVEST@MAIL.MCGILL.CA
Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, CP 6128 succursale Centre Ville, Montréal, QC H3C 3J7 CANADA

Doina Precup DPRECUP@CS.MCGILL.CA
School of Computer Science, McGill University, 3480 University st., Montreal, QC H3A 2A7 CANADA

Abstract

Using neural networks to represent value functions in reinforcement learning algorithms often involves a lot of work in hand-crafting the network structure, and tuning the learning parameters. In this paper, we explore the potential of using constructive neural networks in reinforcement learning. Constructive neural network methods are appealing because they can build the network structure based on the data that needs to be represented. To our knowledge, such algorithms have not been used in reinforcement learning. A major issue is that constructive algorithms often work in batch mode, while many reinforcement learning algorithms work on-line. We use a cache to accumulate data, then use a variant of cascade correlation to update the value function. Preliminary results on the game of Tic-Tac-Toe show the potential of this new algorithm, compared to using static feed-forward neural networks trained with backpropagation.

1. Introduction

One of the important questions in reinforcement learning (RL) research regards the successful use of non-linear function approximators, such as neural networks, in combination with temporal-difference (TD) learning algorithms (Sutton & Barto, 1998). From a theoretical point of view, it is known that some reinforcement learning algorithms oscillate or even diverge with particular non-linear approximators (e.g., Baird, 1995; Tsitsiklis & Van Roy, 1996). On the other hand, some of the most successful applications of RL to large scale tasks, such as the elevator dispatching system (Crites & Barto, 1996) and TD-Gammon (Tesauro, 1995) use TD-learning together with neural networks. In both of these cases, a significant amount of engineering went into setting up the configuration of the neural network, choosing the input parameters and tuning the learning parameters. A lot of anecdotal evidence supports the fact that getting RL to work well with artificial neural networks involves a significant amount of hand-crafting.

In this paper we explore the possibility of using RL algorithms together with constructive neural network algorithms. Constructive neural networks have the desirable property of being able to build the necessary network architecture by themselves, based on the training data provided; hence, such algorithms require less engineering of the network representation and are easier to use. Several empirical studies using supervised learning tasks (e.g., Yang & Honavar, 1998) showed that constructive algorithms perform at least as well or better than hand-tuned feed-forward backpropagation networks.

Although static feed-forward neural networks are often used with RL algorithms, to our knowledge, there have been no published results combining constructive neural networks and RL. The closest attempt is the work of Anderson (1993), combining Q-learning with resource-allocation networks (Platt, 1991). However, in his algorithm the network structure was fixed from the beginning, and hidden units were re-started rather than being added dynamically. A possible reason for this lack of algorithms is that constructive algorithms generally operate in batch mode, over a whole data set, while RL algorithms typically use on-line training. In this paper, we use a variant of the cascade correlation algorithm (Fahlman & Lebiere, 1990; Baluja & Fahlman, 1994), in combination with TD-learning. The neural network is treated like a slow memory, with a cache attached to it. This allows data from the TD-learning algorithm to accumulate in the cache before being used to train the network further. We compare the performance of the constructive algorithm to that of a static backpropagation network in a simple game playing task, that of playing Tic-Tac-Toe. The learning algorithms are tested against different fixed opponents, and different training regimens.

The paper is organized as follows. In Section 2 we present background on the use of reinforcement learning with neural networks. Section 3 describes cached training of neural networks, and the constructive algorithm we are proposing is described in Section 4. In Section 5 we describe the Tic-Tac-Toe domain used as an illustration.

Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003), Washington DC, 2003.

Sections 6 and 7 contain a presentation and discussion of the empirical results. In Section 8 we conclude and present avenues for future work.

2. TD Learning and Neural Networks

Temporal-difference (TD) learning (Sutton, 1988; Sutton & Barto, 1998) is an algorithm used to learn a value function $V^{\pi}: \mathrm{S} \rightarrow \Re$ over the state space S, for a given policy (way of choosing actions) π. In an episodic task (i.e., a task that terminates), within an episode, TD-learning works as follows:

1- Initialize state s.
2- Choose action a using policy π and observe reward r and next state s'.
3- Update $V(s)$ such that $V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$, where α is the learning rate and γ the discount factor (both between 0 and 1).
4- $s \leftarrow s'$
5- Repeat steps 2-4 until episode ends.

The value function learned this way is an estimate of the expected total reward received for an episode from a given state s under policy π.

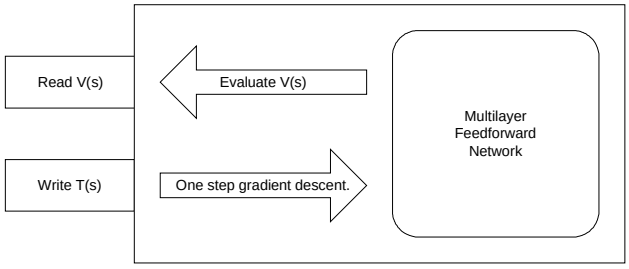

Figure 1. On-line backpropagation network training model.

The usual way to combine TD with neural networks is to represent the value function V^{π} using a multi-layer feed-forward neural network. The state s is described as a vector of input features. The update rule from step 3 above is used to compute the new desired value for state s, denoted $T(s) = (1-\alpha)V(s) + \alpha(r + \gamma V(s'))$. This will be used as target value for state s to generate an error signal $E = [T(s) - V(s)]^2$ for the neural networks. The weights of the neural network are modified such as to minimize this error. Typically, this step is achieved by simply applying gradient descent once. So using the derivative of the error of the network with respect to the weights, these are updated using $W = W - \partial E / \partial W\big|_s$. Note that no learning rate is necessary here, because it was already incorporated in the TD target. Typically, no momentum is used either, since it may interfere with TD bootstrapping. This model of updating, which we will call OnlineBP (for on-line backpropagation) is illustrated in Figure 1. We assume at this point that the network architecture is static, and designed at the beginning of learning. The learning algorithm will change only the network parameters.

3. Cached network training

Because our goal is to use constructive neural network algorithms, we need a mechanism to accumulate patterns without losing the bootstrap effect of on-line learning. We use a simple approach, in which the neural network is treated as a slow memory and a look-up table is used as a cache on it. When the estimated value $V(s)$ for a state s needs to be read, the system first looks it up in the cache. If state s is found, then its cached value is returned. If s is not found, then the network is evaluated for s and the value is saved in the cache before being returned. When a new target value $T(s)$ is computed for a state s, it overwrites the current estimate $V(s)$ in the cache. Hence, the cache always contains a single value $V(s)$ for each state s on which a request was made. This helps to keep the cache small. Also, note that if a state is revisited, its cached value will be updated multiple times, according to the TD update rule. Once in a while, the system needs to be consolidated, by training the neural network on the cache data. In this case, all the values for the different states that appear in the cache are used as targets for the network. Once the system is consolidated, the cache is emptied. This model is illustrated in Figure 2.

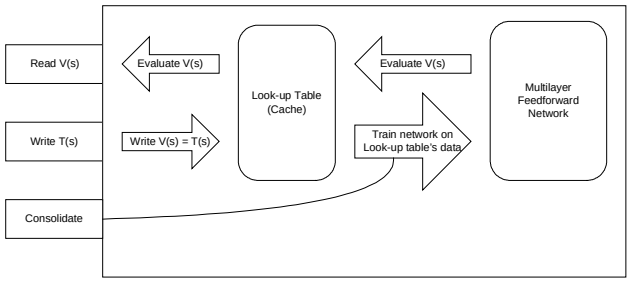

Figure 2. Cached neural network.

Of course, the idea of the cache can be used with static neural networks trained with backpropagation, as well as with constructive algorithms. In the experiments, we use a cache version of backpropagation in addition to the on-line algorithm described above.

4. Constructive Neural Network Learning

The constructive learning algorithm that we use in this paper is a variation of the Cascade-Correlation (CC) algorithm developed by Fahlman and Lebiere (1990). It uses the sibling-descendant pool of candidates suggested by Baluja and Fahlman (1994), which allows a network to grow by adding nodes to the same layer, as well as by adding more layers. This approach allows networks to be less deep than standard CC. We also use a generalization of its candidate objective function (Rivest & Shultz, 2002).

Like CC, the network starts with only inputs and outputs, which are fully connected. The algorithm starts in output phase, in which the weights connecting the output units (the output weights) are trained to minimize the sum of squared error (as in backpropagation). This phase stops when a required accuracy is reached. Exactly like in CC, if the optimization process stagnates, or after a maximum number of epochs, the algorithm shifts to the input phase. The first step of the input phase is to initialize a pool of candidate units that could potentially be added to the network. In standard CC, the pool of candidates consists solely of hidden units that could be cascaded before the output of the network (called descendant units), hence creating a new layer. Instead of this approach, we use the variation of Baluja and Fahlman (1994), which also allows candidate units that could be connected in parallel to the current top-most hidden unit, on the same layer (called sibling units). The weights connecting candidate units are trained to minimize the square of the covariance between their activation and the output residual error normalized by the sum of squared error, i.e. $C_c = Cov(V_c, E)^2 / \sum E_p$. As in CC, at the end of the input phase, which is reached when the optimization process stagnates or after a maximum number of epochs, the best candidate is kept and installed into the network by connecting it to the output units. This candidate will be trained in the next output phase, together with the other output weights. Only output weights are trained in the output phase; the other weights in the network are frozen. In the input phase, descendant candidates that complicate the network architecture are penalized by a given factor (the score C_c of candidate units to be placed on a new layer is multiplied by 80%).

As described in Section 3, a cache is used in addition to the neural network, and the CC-style training uses all the data stored in the cache. The cache targets are computed by TD-learning, as described in Section 2.

5. An illustration: Tic-Tac-Toe

In order to assess the potential of using constructive neural network learning in combination with TD-learning, we experimented with the algorithm described above, and with several other neural network algorithms, in the context of learning to play Tic-Tac-Toe. The main reason for choosing this domain is that it was fairly simple to have automated opponents of different levels of competence. The value function being learned in this case is defined on after-states (Sutton & Barto, 1998). Given the board state s and the possible actions $\{a_1, \ldots, a_n\}$, in order to choose an action, the player evaluates the possible board states s_i resulting from each action (also called after-states). The move to make is selected based on their values. Once the TD player has selected an action, the opponent takes a turn. An update to the value function happens only after the opponent has played and the learning player has chosen its next action, and landed in a new after-state s_i'. Then, the value of the previous after-state, s_i, is updated based on the value of s_i', using usual TD-learning.

An episode is a single game and a non-zero reward is given at the end of the game only. Rewards are -1, 0 and 1 for a loss, a tie and a win respectively. The learning player always plays first. The state of the player is described by 9 features, which constitute the inputs to all the learning algorithms. Each feature corresponds to a location on the board, and its value is 1, if the player occupies the corresponding location, -1, if the opponent occupies the location, and 0 if the location is empty.

The discount factor was set to $\gamma = 1.0$ (meaning that the length of the game does not matter). The behavior of the player is generated using an ε-Greedy policy, which consists of taking the best action based on $V(s_i)$ with probability $(1-\varepsilon)$, but selecting an action randomly with probability ε. The probability of choosing a move randomly was set to $\varepsilon = 0.01$, which means that the learning players make a random move roughly once in 20 games. In preliminary experiments, we tried several values of $\varepsilon \in \{0.01, 0.05, 0.1\}$ but higher values bounded performance. We also tried decreasing exploration schedules, but for the sake of simplicity we are not reporting them here.

We experimented with three learning algorithms: on-line backpropagation, as described in section 2 (*OnlineBP*), batch backpropagation using a cache (*CachedBP*), and the constructive algorithm described in Section 4 (*CachedCC*). Based on some of the results obtained with the constructive algorithm in preliminary trials, in order to make all architectures comparable, the static neural network (used with both on-line and cached backpropagation) was configured to have 9 inputs, followed by 20 layers of 7 symmetric sigmoidal units each (with activation function $f(x) = 1/(1+e^{-x}) - 0.5$). The network is fully connected (every unit receives input from every unit on the previous layer, like in CC). There is a single symmetric sigmoidal output unit with range $[-1.0, \ 1.0]$. This configuration makes the static network structure of the two BP algorithms and the final network structure of the constructive algorithm as close as possible (the final CachedCC networks had around 140 hidden units distributed on an average of 20 layers). Single layer backpropagation networks were also tested with different numbers of hidden units (140, 210, 280, and 315), but none achieved the performance of the topology described above.

For all the learning algorithms, the cache size was set to hold 10 games before consolidating the system. This setting is used in order to keep the cache relatively small, an important requirement for using the algorithm in large state spaces.

The consolidation process (network training) for the *CachedBP* algorithm stops when target values are learned

with 2.5% precision or after 10 epochs. Although 10 epochs may seem low, trials using larger number of epochs (100, 33, and 25, which leads to a total number of epochs similar to CachedCC) showed no significant difference in performance. Moreover, using 10 epochs makes this algorithm comparable to the other two learners in terms of computation time during training. Note that even though CachedBP may seem to require more epochs to reach the CachedCC level, CachedBP updates the hidden weights in each epoch, while CachedCC updates a hidden unit only for a few epochs before freezing it.

For the constructive learning algorithm, the consolidation process (network training) stops when the target values are learned with 2.5% precision ($pr = .05$) (no maximum epoch is required). In preliminary experiments, we tried precision values $pr \in \{.01,.025,.05,.075,.1\}$. Higher values often led to significantly slower learning while lower values often led to bigger variance in the performance of the learned policy, as well as a larger topology of the final network. We suspect that these results are due to overtraining occurring at the lower settings of the precision. Otherwise, default parameter values were used (see Shultz & Rivest, 2001).

Three opponents were set up for the training and evaluation of the learning players: *RandomPlayer* (RP), *BasicPlayer* (BP) and *MinimaxPlayer* (MP). The random player simply chooses a move randomly with uniform probability from the list of available moves. The basic player is equivalent to a myopic search player. If there is a move that would win the game, then the player takes it immediately. Otherwise, if the opponent would have a winning move in the next turn, the position is blocked. If neither of these situations applies, the player chooses an action randomly. The minimax player uses minimax search with alpha-beta pruning and complete depth (i.e., search continues until a final position is reached). The minimax player is therefore perfect and can only win or tie. Its move list is randomized before searching so that it varies its moves.

Table 1. Built-in players' performance against each other in percent wins (mean ± standard deviation).

% WINS		2nd		
		Random	Basic	Minimax
1st	Random	58%±5%	9%±3%	0%±0%
	Basic	88%±4%	31%±5%	0%±0%
	Minimax	97%±2%	77%±4%	0%±0%

Table 2. Built-in players' performance against each other in percent looses (mean ± standard deviation).

% LOSSES		2nd		
		Random	Basic	Minimax
1st	Random	30%±4%	69%±5%	78±4%
	Basic	2%±1%	18%±4%	15±4%
	Minimax	0%±0%	0%±0%	0%±0%

The three opponent players were evaluated by playing against each other 100 games 30 times. This leads to the average %wins and %losses shown in Tables 1 and 2.

For the learning players, 30 instances of each model were trained under each of the conditions listed below:

```
1-  10000 games against RP.
2-  10000 games against BP.
3-  10000 games against MP.
4-  10000 games against a new randomly
    selected opponent for each game.
5-  3000 games against RP, 3000 games
    against BP, 3000 games against MP,
    1000 games against a new randomly
    selected opponent for each game.
6-  3000 games against MP, 3000 games
    against BP, 3000 games against RP,
    1000 games against a new randomly
    selected opponent for each game.
```

Each learning player was evaluated by playing 100 games against each opponent in non-learning mode (using the current greedy policy) before training and after each 1000 training games. Conditions 1, 2 and 3 are aimed at comparing the performance of the different learning algorithms against given, fixed players. Conditions 4, 5 and 6 test the influence of the training regime on the performance of the players. In this case, the learners have to learn to do well against all players, which is a harder task. Under condition 5, one would still expect the learners to do well, since the opponents are presented in increasing order of difficulty. The order is reversed in Condition 6, which intuitively should make the task more difficult.

6. Results

Table 3 contains the mean (and standard deviation) of the percentage of games lost against each type of opponent (as evaluated at the end of the training), for each algorithm under training conditions 1-4. Results are grouped by training condition and results for each algorithm are provided in order.

Table 3. Performance against each built-in player under each training condition in percent lost. Means that are significantly different using the Scheffe test for the Mix condition are starred.

TRAINED AGAINST	MODEL	EVALUATED AGAINST					
		Random		Basic		Minimax	
		Mean	StdDev	Mean	StdDev	Mean	StdDev
Random	Online BP	14%	7%	61%	12%	84%	18%
	Cached BP	9%	5%	57%	7%	97%	8%
	Cached CC	4%	4%	34%	17%	56%	31%
Basic	Online BP	9%	3%	54%	6%	100%	0%
	Cached BP	9%	3%	55%	5%	100%	0%
	Cached CC	8%	5%	17%	10%	37%	41%
Minimax	Online BP	17%	5%	43%	14%	50%	32%
	Cached BP	20%	5%	41%	6%	29%	9%
	Cached CC	18%	9%	39%	15%	17%	24%
Mix	Online BP	6%	3%	37%	14%	82%*	21%
	Cached BP	8%*	4%	46%*	17%	92%*	19%
	Cached CC	5%*	4%	29%*	21%	29%*	33%

It is clear from this table that CachedCC is either as good as the other learning algorithms, or it outperforms them. Its loss percentage is always equal or below that of the other algorithms for every condition (except against RP under MP training, but this is clearly not a significant difference, so it can be considered equal). To control the significance of these results, a one-way ANOVA analysis was done on the MIX condition performance against each type of opponent independently. A post-hoc Scheffe test shows that CachedCC is significantly better (at the .05 level) than CachedBP against every opponent independently. It also shows that CachedCC is significantly better than OnlineBP against the MP opponent.

Table 4 contains the mean (and standard deviation) of the percent of wins against each type of opponent at the end of the training, for each algorithm, under training conditions 1 to 4. Here, CachedCC is outperformed by OnlineBP once (against BP under MIX training). But in that condition, CachedBP also performs worse, which seems to indicate that the caching mechanism may be partially the cause. To validate the significance of these results, a one-way ANOVA analysis was done on the MIX condition performance against each type of opponent independently. A post-hoc Scheffe test confirmed that CachedCC is significantly worse (at the .05 level) than OnlineBP against the BP opponent. CachedBP is not significantly different than the two others. Nevertheless, CachedCC is either as good as the other algorithms or it outperforms them in every other condition.

Table 4. Performance against each built-in player under each training condition in percentage of games won. Means that are significantly different using Scheffe test for the Mix condition are starred.

TRAINED AGAINST	MODEL	EVALUATED AGAINST					
		Random		Basic		Minimax	
		Mean	StdDev	Mean	StdDev	Mean	StdDev
Random	Online BP	78%	11%	20%	12%	0%	0%
	Cached BP	87%	5%	36%	11%	0%	0%
	Cached CC	87%	7%	26%	15%	0%	0%
Basic	Online BP	80%	5%	23%	4%	0%	0%
	Cached BP	81%	4%	23%	4%	0%	0%
	Cached CC	82%	6%	32%	14%	0%	0%
Minimax	Online BP	57%	11%	12%	6%	0%	0%
	Cached BP	52%	5%	14%	4%	0%	0%
	Cached CC	69%	10%	12%	8%	0%	0%
Mix	Online BP	87%	4%	37%*	13%	0%	0%
	Cached BP	86%	4%	30%	16%	0%	0%
	Cached CC	87%	6%	27%*	16%	0%	0%

With every learning algorithm, when trained against a fixed opponent, a few networks learned to never lose against that particular opponent. But when trained under the mix or sequential conditions, only a few CachedCC networks learned to never loose against any opponent. Results for the networks that never lose against BP and MP are in Table 5. There were four such networks in the mix condition 4, four in the sequence condition 5, and one in the inverse sequence condition 6.

Table 5. Results for the Cached CC networks that never lose against BP and MP.

	%Lost vs RP	%Win vs RP	%Win vs BP
Mix	½%± ½%	94%±3%	40%±12%
Sequence	3½%±½%	90%±1%	26%±10%
Inv. Seq.	2%	92%	35%

Table 6 contains the mean (and standard deviation) of the percentage of losses against each type of opponent at the end of the training for each algorithm under training condition 4, 5, and 6. Results are grouped by algorithm. Each algorithm shows some decrease in performance in the inverse sequence condition compared to the mix condition. Although OnlineBP is only negatively affected in the inverse sequence condition, the algorithms using the cache are affected in various ways. CachedBP is doing slightly worst against RP (significantly at the .05 level using independent sample t-test) and it is doing much better against MP in the incremental setup (significantly at the .05 level using independent sample t-test). CachedCC is doing better (but not significantly better) against each opponent in the sequence condition. Also note that CachedCC is the only algorithm to reach its independent training performance: even though only around 3333 games are played against each opponent, some of the networks get to the level of play achieved by playing 10000 games against that opponent (see Table 3).

Table 6. Performance against each built-in player under each training condition in percentage games lost. Pairs of values compared with t-test are marked by a star if significantly different and a hat if not.

MODEL	TRAINED AGAINST	EVALUATED AGAINST					
		Random		Basic		Minimax	
		Mean	StdDev	Mean	StdDev	Mean	StdDev
Online BP	Mix	6%	3%	37%*	14%	82%*	21%
	Sequence	7%	5%	42%	16%	84%	23%
	Inv. Seq.	8%	4%	47%*	15%	93%*	14%
Cached BP	Mix	8%*	4%	46%*	17%	92%*	19%
	Sequence	15%*	4%	43%	9%	45%*	22%
	Inv. Seq.	9%	3%	57%*	11%	100%	0%
Cached CC	Mix	5%*	4%	29%*	21%	29%^	33%
	Sequence	5%	3%	18%*	14%	19%^	29%
	Inv. Seq.	9%*	6%	35%	17%	36%	33%

Table 7 contains the mean (and standard deviation) of the percentage of games won against each type of opponent at the end of the training for each algorithm under training conditions 4 and 5. Results are grouped by algorithm. Here, OnlineBP is affected by the reverse sequence condition. CachedBP is clearly negatively affected (t-test were done for BP and RP and both were significantly different at the .05 level) in the sequence condition and, surprisingly, unaffected by the inverse sequence condition. CachedCC is unaffected and is also, again the

only algorithm that allows some learners to reach the independent training condition level.

Table 7. Performance against each built-in player under each training condition in percentage games won. Pairs of values compared with t-test are marked by a star if significantly different and a hat if not.

MODEL	TRAINED AGAINST	EVALUATED AGAINST					
		Random		Basic		Minimax	
		Mean	StdDev	Mean	StdDev	Mean	StdDev
Online BP	Mix	87%	4%	37%	13%	0%	0%
	Sequence	86%	6%	31%	15%	0%	0%
	Inv. Seq.	86%	4%	35%	13%	0%	0%
Cached BP	Mix	86%*	4%	30%*	16%	0%	0%
	Sequence	61%*	8%	12%*	8%	0%	0%
	Inv. Seq.	85%	4%	28%	10%	0%	0%
Cached CC	Mix	87%	6%	27%	16%	0%	0%
	Sequence	85%	6%	25%	14%	0%	0%
	Inv. Seq.	83%	8%	26%	14%	0%	0%

Overall, each algorithm is doing worst in the reverse sequence, but it seems that OnlineBP is unaffected by the normal sequence. CachedBP is affected and an analysis of its learning curves shows that cached BP is strongly biased toward most recent training. Figure 3, Figure 4 and Figure 5 show that the performance of CachedBP against RP and BP decreases when learning to play against MP; hence, CachedBP seems to show an important forgetting effect. On the other hand, playing solely against MP in the sequential condition gives it an advantage over MP that it does not have in the mix condition, as shown in Figure 6 and Figure 7.

This may also explain why CachedCC is only playing slightly better under the incremental condition. Its constructive approach along with the incremental learning suggests that we should get much better results in the incremental setup; instead, we only get a small increase in performance and a huge variation. This may be due to the overall worse performance of algorithms that use the cache. Nevertheless, CachedCC is clearly less sensitive to catastrophic forgetting than CachedBP, certainly due to its weight freezing.

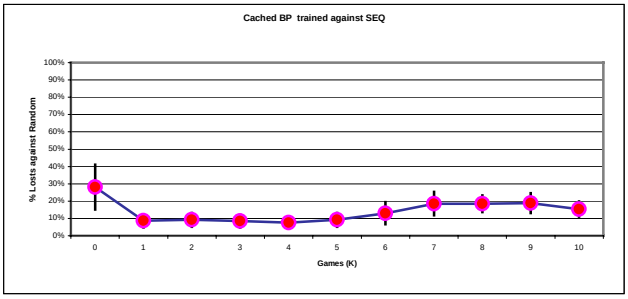

Figure 3. Percent lost of Cached BP against RP trained incrementally. It shows a small forgetting of RP when learning against BP (4 to 6) and MP (7 to 9).

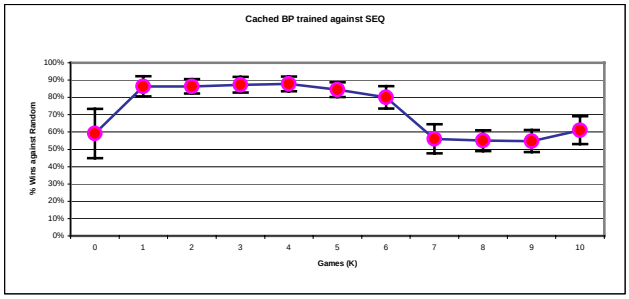

Figure 4. Percent win of Cached BP against RP trained incrementally. It shows a small forgetting of RP when learning against BP (4 to 6) and MP (7 to 9).

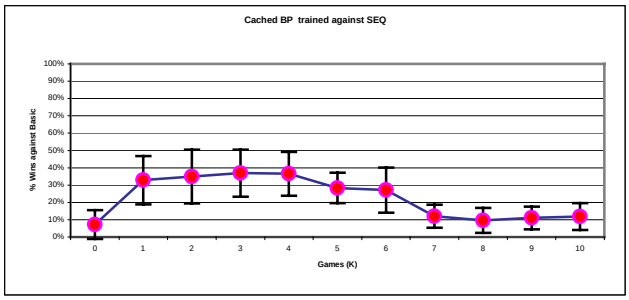

Figure 5. Percent win of Cached BP against BP trained incrementally. It shows a small forgetting of BP when learning against MP (7 to 9).

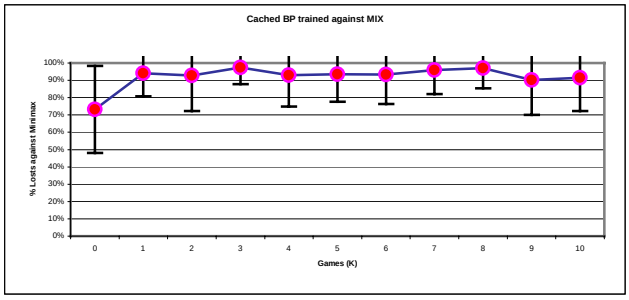

Figure 6. Percent lost of Cached BP against MP trained in mix condition. It does not improve successfully against MP.

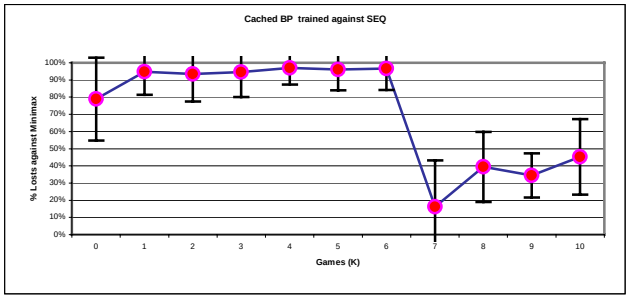

Figure 7. Percent lost of CachedBP against MP trained incrementally. It improves only near the end when learning against MP itself (7 to 9).

7. Discussion

There are few interesting things to note about the constructive approach and how the networks grow under the various training conditions. Considering how poorly a learner trained against RP or BP only performs against MP, one would expect systems trained against a sequence of opponents of increasing difficulty to show a sudden jump in size when they start playing against a new opponent. However, this does not happen at all in our experiments, as shown in Figure 8. In fact, such size increases only happen in the inverse sequence condition (Figure 9). Structural graphs show that, in fact, the structure developed while playing against RP is almost sufficient to learn to play against BP and MP, even though the non-output weights of the network are frozen. More interestingly, the structural curves under the randomly selected opponent condition show a slower increase in the number of nodes, over a longer period of time, than the curves trained with a sequence of increasing difficulty, as shown in Figure 10 and Figure 8. The latter exhibit a jump in the beginning, then more or less stagnate for the rest of the training.

Figure 10 and Figure 11 show growth curves in number of hidden units and in number of layers (network depth) respectively.

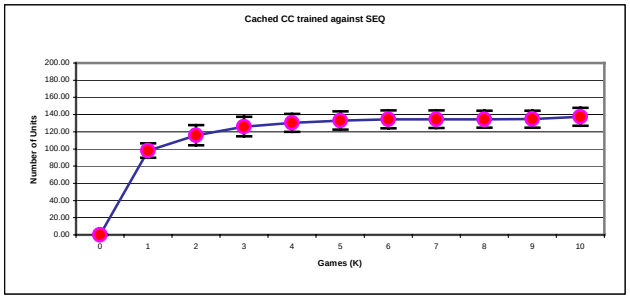

Figure 8. CachedCC network growth curve while learning in the sequence condition.

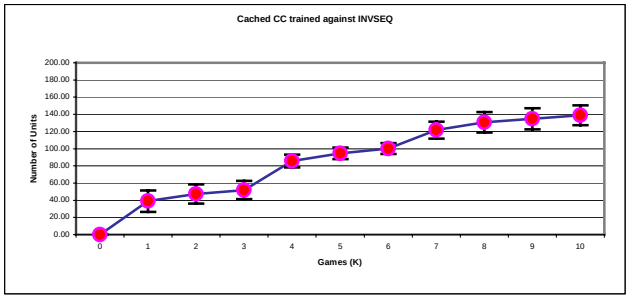

Figure 9. CachedCC network growth curve while learning in the reverse sequence condition.

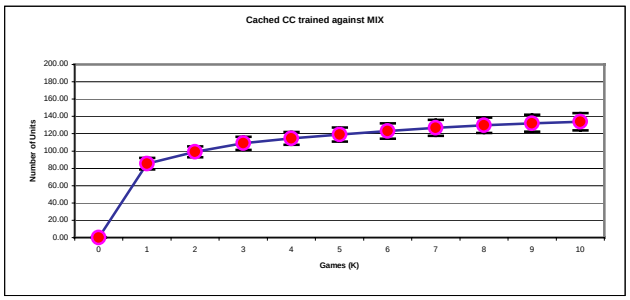

Figure 10. CachedCC network growth curve while learning in the mix condition.

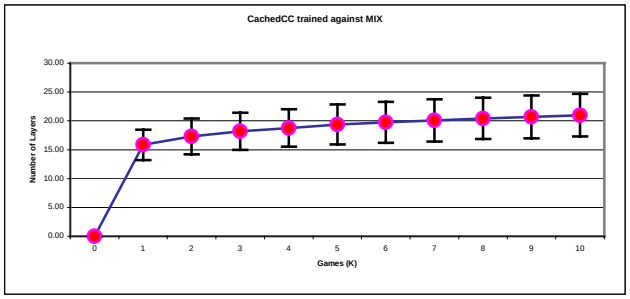

Figure 11. CachedCC network depth curve while learning in the mix condition.

It is also interesting to note that slight variations in the behavior of the cache can change the performance of the learners significantly. In the model used, every time a value $V(s)$ is read, it is placed in the cache. When the agent selects its move using its current policy, it looks at every possible move, and hence, at multiple after-states. From these, only the after-state based on the move chosen will be updated (or corrected). A different variation is to have the cache hold only the values that need to be updated. But in preliminary trials, this approach led to less successful results. One of the reasons is certainly the natural rehearsal that the read cache places in the training set. That is, the training set does not only contain patterns that need correction, but also patterns that do not need to change. This restricts the way in which the network may

change, and has the effect of avoiding catastrophic forgetting of the current knowledge (which could result from learning only corrections).

This kind of rehearsing has already been successfully used in transfer of knowledge in neural networks in the task rehearsal method (Silver, 2000). It was also used by Ans & Rousset (2000) in their self-refreshing memory, a model in which two neural networks used rehearsing to learn multiple tasks one after the other, while reducing catastrophic forgetting.

In a similar fashion, if the player is doing some search before choosing its move, like in the TD-Leaf algorithm (Baxter, Tridgell, & Weaver, 1998), it is possible to place search information into the cache, in order to bootstrap the learning prior to taking real actions. This may help generate a richer training set for the network.

8. Conclusion

In this paper we explored the use of constructive neural networks with reinforcement learning algorithms. The particular algorithm we used is based on Cascade-Correlation, and uses a short-term look up table as cache to accumulate a batch of data. Our empirical results indicate a strong potential for this combination. The algorithm did not exhibit any catastrophic forgetting, like batch backpropagation. Also, the constructive algorithm was the only one to reach the same performance when trained against a mix of opponents, as when training took place against one opponent only. Of course, a lot more experimentation is needed to assess the merits of this approach, in particular given that we observed some variance in its performance under different conditions.

Acknowledgements

This research was supported in parts by grants from NSERC and FQNRT. We also want to thank our anonymous reviewers for their very helpful comments.

References

Anderson, C.W. (1993) Q-Learning with Hidden-Unit Restarting. In *Advances in Neural Information Processing Systems 5*, pp. 81-88. MIT Press.

Ans, B. & Rousset, S. (2000) Neural Networks with a Self-Refreshing Memory: Knowledge Transfer in Sequential Learning Tasks Without Catastrophic Forgetting. *Connection Science* 12(1):1-19.

Baird, L.C. (1995). Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 30-37. Morgan Kaufmann, San Francisco.

Baluja, S., & Fahlman, S.E. (1994). *Reducing Network Depth in the Cascade-Correlation Learning Architecture.* Technical Report #CMU-CS-94-209, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Baxter, J., Tridgell, A., & Weaver, L. (1998) TDLeaf(lambda): Combining Temporal Difference Learning with Game-Tree Search. In *Proceedings of the Ninth Australian Conference on Neural Networks*, pp. 168-172, Brisbane QLD.

Crites, R.H. & Barto, A.G. (1996). Improving elevator performance using reinforcement learning. In D.S. Touretzky, M.C. Mozer & M.E. Hasselmo (eds.), *Advances in Neural Information Processing Systems: Proceedings of the 1995 Conference*, pp. 1017-1023. MIT Press, Cambridge MA.

Fahlman, S.E. & Lebiere, C. (1990) The cascade-correlation learning architecture. In *Advances in neural information processing systems 2*, pp. 524-532. Los Altos, CA: Morgan Kaufmann.

Platt, J.C. (1991) A resource-allocating network for function interpolation. *Neural Computation* 3:213-225.

Rivest, F., & Shultz, T.R. (2002). Application of Knowledge-based Cascade-Correlation to Vowel Recognition. In *Proceedings of the 2002 International Joint Conference on Neural Networks*, pp. 53-58.

Silver, D. (2000*). Selective Transfer of Neural Network Task Knowledge.* Ph.D. Thesis, University of Western Ontario.

Shultz, T.R. & Rivest, F. (2001) Knowledge-based cascade-correlation: Using knowledge to speed learning. *Connection Science* 13:1-30.

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

Sutton, R.S., & Barto, A.G. (1998). *Reinforcement Learning: An Introduction*. The MIT Press: Cambridge, Massachusetts. 340 p.

Tesauro, G.J. (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM 38*, pp. 58-68.

Tsitsiklis, J.N. & Van Roy, B. (1996). Feature-based methods for large scale dynamic programming. *Machine Learning*, 22:59-94.

Yang, J. & Honavar, V. (1998) Experiments with the cascade-correlation algorithm. *Microcomputers Applications*, 17:40-46.