Cascade correlation algorithms for on-line reinforcement learning

Keywords: Temporal-difference learning, function approximation, cascade correlation
Marc Gendron-Bellemare MGENDR12@CS.MCGILL.CA
School of Computer Science, McGill University, 3480 University st. Montreal, QC, H3A2A7, CANADA

Abstract

Neural networks have been used very successfully to represent value functions for reinforcement learning algorithms in several applications. However, considerable hand-crafting is needed in order to obtain a good network structure. To address this problem, (Rivest & Precup, 2003) proposed an approach to using cascade correlation neural networks with reinforcement learning. Their approach relies on a cache of data, which is used to accumulate a batch of patterns for training the network. In this paper, we explore a version of cascade correlation which has a more on-line flavor. Empirical results on the game of Tic-Tac-Toe show that the variations we discuss are more robust to the noise inherent in RL tasks.

1. Introduction

Reinforcement learning (RL) applications often require the use of function approximators, such as linear function approximators or neural networks, in order to represent value functions (Sutton & Barto, 1998). Neural networks, in particular, have a history of successful applications in reinforcement learning (e.g. Crites & Barto, 1996; Tesauro, 1995). However, in order to get neural networks to perform well with RL algorithms, a significant amount of engineering is typically required, in terms of choosing the input representation, the configuration of the neural network and the learning parameters.

In order to address the problem of determining a good network structure, Rivest & Precup (2003) proposed a mechanism for using cascade correlation neural networks with temporal difference (TD) learning (Sutton, 1988). Cascade-correlation networks (Fahlman & Lebiere, 1990; Baluja & Fahlman, 1994) construct a network topology automatically, using a batch of data. However, RL algorithms typically work on-line, updating the approximation of the value function incrementally based on every sample. In order to address this mismatch, Rivest & Precup proposed to use a cache of samples, working as a table, in order to create a batch of data for the neural network. Once the cache is full, the network is trained on this data and allowed to change its configuration. However, the use of the cache, as described in this work, poses three problems. First, the size of the cache has a significant influence on performance, so this is now a new parameter that has to be set. Second, each state appears in the cache exactly once. If a state is visited many times, its value estimate in the cache will be updated. However, there is no record of the importance of the state relative to other states, and all members of the cache are treated equally when creating new units. This is justified in supervised learning, where each training example is assumed to be drawn independently, and is present in the batch of data exactly once. However, in RL, states are visited with different frequencies; intuitively, states that are visited more frequently are more important, and the learning algorithm should focus more on getting their values correctly. A third issue is that of the criterion used to decide when a new unit has been sufficiently trained. The criterion used by Rivest & Precup requires that all patterns in the cache should be approximated within a given precision, just like in supervised learning. However, in RL, data is often very noisy. As such, this criterion seems too stringent and possibly prone to overfitting, if the precision required is too high.

In this paper we present some variations of cascade-correlation with TD-learning. We experiment with a version of cascade correlation which has an on-line flavor. In order to decide when it is time to add more units, we do not use a cache but instead we monitor the error signal on each time step. Whenever the error stops decreasing, the network is allowed to add more units. At this point, a small batch of data is built in order to train the new units. However, this batch does not act like a cache. Instead, it is just a history of visited states and TD targets for them.

The paper is organized as follows. In Section 2 we re-

Appearing in *Proceedings of the North East Student Colloquium on Artificial Intelligence*, Ithaca, New York, 2006.

view TD-learning with neural networks and with cascade-correlation networks. In Section 3 we present our version of on-line cascade correlation, and an alternate stopping criterion which reduces some overfitting effects present when using a Minkowski-infinity norm criterion. Section 5 studies the empirical performance of the Cascade-correlation algorithms on the game of Tic-Tac-Toe. Finally, Section 6 contains a discussion of the results and avenues for future work.

2. Background

Temporal-Difference (TD) learning (Sutton, 1988; Sutton & Barto, 1998) is a standard algorithm for learning the value function $V^{\pi} : S \rightarrow \Re$ for a fixed, given policy (way of choosing actions) π and where S denotes the problem's state space. On every time step, a reinforcement learning agent observes the state of the environment, s, chooses an action a according to policy π, and observes an immediate reward r and a next state s'. The approximate value function $V(s)$ is updated based on this data as follows:

$$V(s) \leftarrow V(s) + \alpha(r + \gamma V(s') - V(s))$$

where $\gamma \in [0,1]$ is a discount factor used to weight less rewards received later in the future, and $\alpha \in (0,1)$ is the learning rate.

When TD learning is combined with neural networks, the state s is represented as a vector of features, and the value function $V(s)$ is represented as a neural network. After each transition, a new desired target for state s is computed as above: $T(s) = V(s) + \alpha(r + \gamma V(s') - V(s))$. In the case of a network with fixed configuration, the weights are then trained to minimize the squared error, $E = [T(s) - V(s)]^2$, by stochastic gradient descent.

Constructive neural networks, which build the network structure over time, typically need a batch of data in order to find a good configuration. In order to build such a batch, Rivest & Precup (2003) proposed using a cache at the interface between the environment and the neural network. Whenever a state s is first encountered, it is stored in the cache, together with its desired target. If the state s is already in the cache, its desired target value is updated using the TD update rule, as if the cache were a table. This choice was made in order to keep the cache small. When the cache is full, the data is used as a batch for training. The main disadvantage of this implementation is that each pattern in the cache occurs exactly once. Even if some states are visited often, this information is not present in the cache.

Determining the size of the cache is also an important issue. In practical applications, an agent would not really follow a fixed policy π, but instead would pick moves mostly greedily based on its current value estimates (Sutton & Barto, 1998). If this is the case, a large cache may have data about states which were visited under previous policies, but which are no longer relevant. On the other hand, if the cache is too small, units will be added very frequently and the network will grow too large.

3. Online Cascade-Correlation

We propose a version of cascade correlation which is closer to an on-line algorithm. Our work builds on the ideas presented in (Rivest & Precup, 2003), but avoids working with the cache. Like (Rivest & Precup, 2003), we work with the cascade-correlation algorithm introduced in (Fahlman & Lebiere, 1990) and further refined in (Baluja & Fahlman, 1994).

Like in cascade correlation, the network starts with only input units and output units, which are fully connected. The algorithm starts in output phase, during which the weights feeding into the output units are trained to minimize the squared error (like in standard backpropagation). This step is done online, by performing one update on each state transition, instead of accumulating a batch of data first. In this way, learning should be faster.

Once n patterns have been processed, we begin testing at every new pattern whether the error is still being reduced. If the algorithm detects that further training will not improve the estimation error, it then switches to input phase. We discuss the criteria for switching in the next section.

In the input phase, we initialize a set of candidate units that could potentially be added to the network. In standard cascade correlation, units are added only in a new layer. Instead, like in (Baluja & Fahlman, 1994) and (Rivest & Precup, 2003), we consider adding both siblings to the units in the current topmost layer, as well as units in a new layer. This results in a network topology that is more similar to standard backpropagation neural networks. In the input phase, a batch of data is still necessary, because the weights from the previous units to the new candidates have to be trained such as to correlate the residual error. In order to compute the correlation, a measure of average error is necessary, which can only be obtained from a batch of data. In order to perform this step, we simply gather a batch of N previous transitions. These can be taken from the data used for the previous output phase, or they can be based on new data. Since this is just a history window, states that are revisited during this time will just appear multiple times in the batch. Once the agent has gathered a batch of N data points, we proceed exactly as in (Rivest & Precup, 2003), training the input weights of the candidate units to maximize their correlation with the residual error. Once the process stagnates or enough passes have been done on the data, the candidate that best captures the error is selected and installed in the network. The criterion for detecting whether the input phase should be completed is the same as in (Shultz & Rivest, 2001). In order to favor shallow topologies, we multiply the score of the candidates on the next layer by 0.8, as suggested in (Baluja & Fahlman, 1994). Once installation of the new unit is complete, its input weights are frozen and training resumes in the output phase. At this stage, only the weights between the topmost layer and the output units are trained, in a non-correlation fashion.

Note that any method for valuation can be used in order to produce target values for the algorithm. We chose, for simplicity, to use the basic TD(0) algorithm, discussed above, but this approach can be used with other RL targets as well.

4. Switching between output and input phase

When cascade correlation networks are used in supervised learning, as well as in the previous work on cascade correlation in RL, the criterion typically used to detect whether the output phase should be terminated is based on the Minkowski-infinity norm. More precisely, if the maximum absolute error attained for any pattern in the cache, $\max_s |T(s) - V(s)|$, is larger than a specified threshold t for a long enough period of time, it is deemed that value estimates have ceased to improve and the learning switches to input phase, adding more units. Hence, the networks grow until the error for every pattern encountered is below the given threshold. This is very reasonable for supervised learning, where each data point is assumed to be correct. However, this is less desirable in the case of RL, where we wish to learn quickly and approximately. Intuitively, if the data is very noisy (as is the case in RL) and we require a certain precision on every pattern, then the network would likely grow too large and be prone to overfitting. Rivest & Precup discuss this issue as well, and propose the use of a large threshold t to avoid this problem, but in general it may be hard to tell how this threshold should be chosen for an arbitrary task. In our work, we used a different criterion which intuitively seems better suited for RL.

A standard way of measuring whether the algorithm is stagnating is to look at the sum-squared error (L_2 norm) over the training patterns in the window. This is the measure which the weight updates are trying to optimize. If the sum-squared error stays above a threshold, $\sum_i (T(s_i) - V(s_i))^2 > t$ for long enough, then we assume that learning stagnates and allow the algorithm to add new units. Since we are working with a stopping criterion that is closer to what we are attempting to minimize, we should observe improvements over the original Minkowski-infinity norm criterion.

5. Tic-Tac-Toe

(Rivest & Precup, 2003) showed some results on the Cached Cascade-Correlation algorithm on the game of Tic-Tac-Toe. In order to relate to their work, we ran our algorithm on this game as well. We follow the same game representation as they proposed, with modifications where necessary. Assuming that the reader is already familiar with the game itself, we will give a brief overview of how we set up the task in a RL framework.

5.1. Description

The board is described by nine inputs, each of which takes an integer value between -1 and 1, where 0 represents an empty square, -1 an opponent square and 1 a square belonging to the player. Note that such a setup is symmetric from a functional point of view, which intuitively makes sense. The reward is simply of +1 if the agent wins, and -1 if it loses, with no reward in the event of a tie. To simplify matters, the learning agent always plays first.

The value function we are interested in learning uses the concept of after-state, discussed in (Sutton & Barto, 1998). Instead of evaluating the value of the current position, we compute the value of the *resulting* state once the agent has chosen an action. Setting up the problem this way is essential as we do not have a simple way of modelling the transition probabilities from one state to another if the opponent is unknown.

The output phase had a minimum length of 1000 patterns, and the batch of data used for the OC agents in input phase was 100 patterns. In preliminary experiments, the exact size of this batch had little effect on performance. Similarly, a cache of 100 patterns was used for the CC agents. In this set of experiments we use the bias criterion for switching from output phase to input phase. We set the phase change threshold to $t = 0.0001$, a value that ensures that no units would be added once the value function has stabilized. This criterion is used both for OC and for CC networks.

5.2. Experimental Setup

Three types of fixed-policy opponents were used for training and testing purposes: a *Random* player (RP), a *Basic* player (BP) and a *Minimax* player (MP). The random player selects actions uniformly randomly from the list of valid actions. The basic player corresponds to the one described in (Rivest & Precup, 2003), and plays with a search depth of 1. Finally, the minimax player follows an optimal (win or tie) policy, found by searching over the entire set of possibilities.

The number of games was kept at 10000 for all experiments; however, we studied the effect of changing the op-

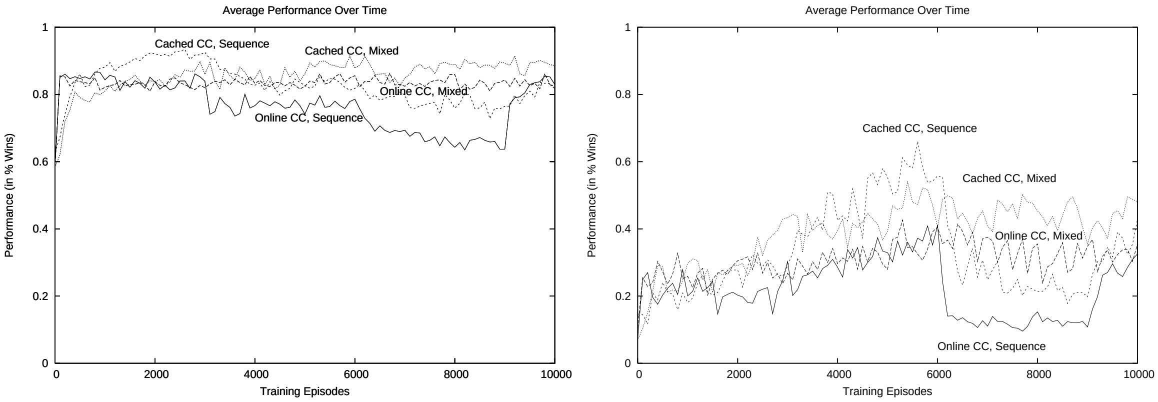

Figure 1. Performance of Tic Tac Toe agents using Cached and Online Cascade-Correlation against the random (left) and basic (right) agents. Both sequence and mixed training regimens are shown. Performance here is given in number of wins.

ponents against which the agents were trained. In the first setup, the agent trained against RP for 3000 games, against BP for 3000 games, then against MP for 3000 games, and finally against a "mix" of all 3 opponents for 1000 games. In the mix condition, a different player is selected at random for each game. In the second setup, the agent trained only in the mix condition.

On one hand, we would expect agents to perform well in the first setup, as they are allowed to learn against weaker opponents before moving on to stronger ones. However, this situation also presents the agent with a non-stationary environment, as opposed to a Markovian one. Hence, this is a good way to test if forgetting occurs.

To evaluate the algorithm's performance, we played each agent for 100 games against the three fixed players, using its greedy policy. This was done before training, and subsequently after every 100 training games. The results, averaged over 10 independent runs, are given in Fig. 1. The OC agents learn very quickly, but are also subject to quick forgetting, as can be seen in the sharp drops in the performance against the random player at time steps 3000 and 6000. The most striking example is when training against the minimax player (Figure 2). The OC agents learn very quickly, and get much better than the other agents. However, this is at the expense of performance loss against the other agents. Once the training regimen is changed to mixed, the performance advantage is also lost. In the mixed regime, OC agents learn faster, but then stabilize at a slightly lower performance level than their CC counterparts.

One of the main reasons as to why forgetting occurs more proeminently in OC than in CC seems to be the number of hidden units added. The earlier hidden units incorporate information about the random and basic agents that are not lost when training against the minimax agent. This would seem to be a point in favor of CC. However, it is also possible that the number of connections to be trained for the output unit becomes much larger, with more local minima and therefore the algorithm has in effect reached its maximum performance. This is problematic in most tasks as the stabilization may happen before the algorithm has found a satisfying solution. This is especially true in reinforcement learning where targets change over time as they converge to the optimal value function. In Tic-Tac-Toe, trajectories through the state space cannot contain cycles. This feature might therefore not be as important here, especially since the branching factor is relatively low. Furthermore, partial forgetting can be a desirable property of a RL algorithm when the environment changes quickly or presents stochasticity. In such cases, overdriving units to respond to very specific features might only increase the complexity of the network, and as discussed above, hinder learning.

Also note that the online algorithm had the most trouble producing a good policy against the minimax player when mix-trained. This could be due to the fact that the agent is attempting to learn three different policies at once.

6. Discussion and future work

In this paper, we presented an online version of the cascade-correlation algorithm, aimed at reinforcement learning. Using a cache can potentially introduce problems such as overfitting and skewing of the state distribution. Our online algorithm partly addresses these issues by using the data obtained from each state transition exactly once. This results in an algorithm that can learn stochastic tasks, but which is prone to similar problems as standard feed-forward neural networks such as forgetting. Using a cache results in learning that is slower but more stable. The results given here and past experiments point towards the idea that constructive neural networks in the context of non-

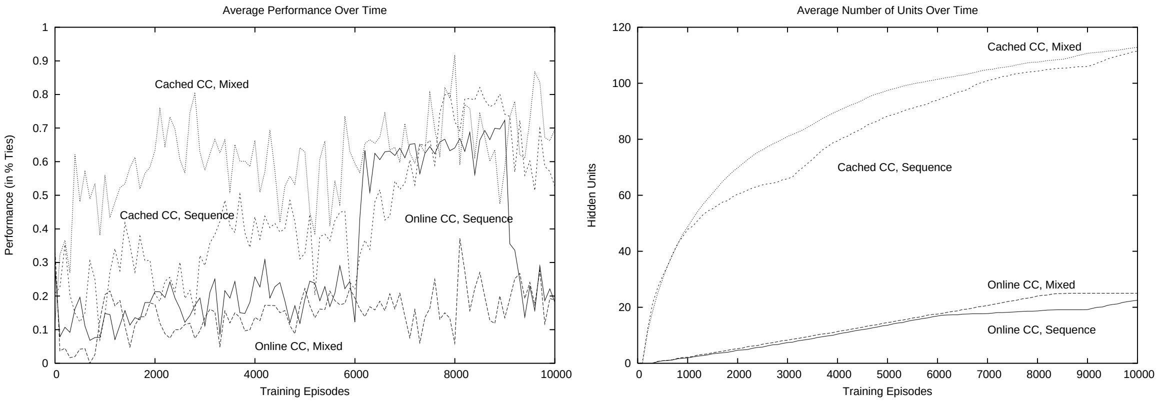

Figure 2. Left: Performance of Tic Tac Toe agents using Cached and Online Cascade-Correlation against the optimal player, in percentage of ties. Right: Number of hidden units in function of time.

stationary data can be very sensitive to unit addition. There is a delicate balance that must be achieved between discovering and adding useful units and preventing the algorithm to converge properly.

There are, however, several unresolved issues. For both types of agents, stabilization of the network size is sensitive to the choice of the threshold parameter that controls when the addition of more units is allowed. Based on previous experiments, we consider that the OC networks are better suited for environments that are highly stochastic and where the value function changes slowly enough. The CC networks are better in the presence of non-stationarity, as they exhibit less forgetting.

At the core the problem is therefore the idea that unit addition should be given a high priority in the learning process. Unfortunately, determining how a new unit should be constructed might require studying the correlation between the already present hidden units, in order to avoid redundancy. One possibility for this would be to add many units at one time, training them to correlate with the error signal but also to be as *independent* as possible from one another. Better measures and criteria for determining when to add units also seems critical.

Acknowldgements

The author would like to acknowledge Doina Precup and Francois Rivest for long hours of discussion on the topic of neural networks and feature construction. We would also like to thank Frederic Dandurand for further insight on the Cascade-Correlation architecture.

References

Baluja, S., & Fahlman, S. (1994). Reducing network depth in the cascade-correlation learning architecture.

Bellemare, M. G., Precup, D., & Rivest, F. (2004). *Reasoning and learning lab: Technical report rl-3.04* (Technical Report). McGill University.

Crites, R. H., & Barto, A. (1996). Improving elevator performance using reinforcement learning. *Advances in Neural Information Processing Systems* (pp. 1017–1023). The MIT Press.

Fahlman, S., & Lebiere, C. (1990). The cascade-correlation learning architecture. *Advances in Neural Information Processing Systems* (pp. 524–532). Denver, 1989: Morgan Kaufmann, San Mateo.

Rivest, F., & Precup, D. (2003). Combining td-learning with cascade-correlation networks. *ICML 2003*.

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

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

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

Tesauro, G. (1995). Temporal difference learning and td-gammon. *Communications of the ACM*, *38*, 58–68.