Global Versus Local Constructive Function Approximation For On-Line Reinforcement Learning

Peter Vamplew and Robert Ollington

Technical Report 2005-01
School of Computing, University of Tasmania, Private Bag 100, Hobart

This report is an extended version of a paper of the same title presented at AI'05: The 18th Australian Joint Conference on Artificial Intelligence, Sydney, Australia, 5-9 Dec 2005.

Abstract: In order to scale to problems with large or continuous state-spaces, reinforcement learning algorithms need to be combined with function approximation techniques. The majority of work on function approximation for reinforcement learning has so far focused either on global function approximation with a static structure (such as multi-layer perceptrons), or on constructive architectures using locally responsive units. The former, whilst achieving some notable successes, has also been shown to fail on some relatively simple tasks. The locally constructive approach has been shown to be more stable, but may scale poorly to higher-dimensional inputs, as it will require a dramatic increase in resources. This paper explores the use of two constructive algorithms using non-locally responsive neurons based on the popular Cascade-Correlation supervised-learning algorithm. The algorithms are applied within the sarsa reinforcement learning algorithm, and their performance compared against both a multi-layer perceptron and a locally constructive algorithm (the Resource Allocating Network) across three reinforcement learning tasks. It is shown that the globally constructive algorithms are less stable, but that on some tasks they can achieve similar performance to the locally constructive approach, whilst generating much more compact solutions.

1 Introduction

Reinforcement learning addresses the problem of an agent interacting with an environment. At each step the agent observes the current state and selects an action. The action is executed and the agent receives a scalar reward. The agent has to learn a mapping from state-to-action to maximise the long-term reward. One way to do this is to learn the expected return, either per state or per state-action pair. Many algorithms for learning these values are based on the use of temporal differences (TD) [1] where the value of the current state at each step is used to update the estimated value of previous states.

For problems with small state-spaces the values can be stored in a table, but as the dimensionality or resolution of the state increases, the storage requirements become impractical. In addition learning slows, as tabular algorithms can only learn about states and actions which the agent has experienced. For these tasks function approximation must be used to estimate the values. This can usually be achieved using far fewer parameters than the number of states thereby reducing storage. In addition function approximators can generalise from states which have been experienced to similar states that are yet to be visited, hence increasing the rate of learning.

Two main approaches to function approximation have been explored in the reinforcement learning literature. One involves the use of global approximators such as neural networks. These have been applied successfully to a range of problems, such as elevator control [2] and backgammon [3]. However this success has failed to be replicated on other, seemingly similar tasks. The second approach is to use locally sensitive approximators such as CMACs [4] or radial-basis functions [5]. The local approach has been shown to be more stable and more amenable to formal analysis, but may scale less well to higher-dimensional input spaces [6].

Several sources have provided arguments for the use of constructive networks for function approximation in reinforcement learning systems. [7] provides both a theoretical argument and empirical evidence that function approximators are prone to systematic overestimation of values when used with existing reinforcement learning algorithms (particularly for off-policy algorithms). Whilst not directly advocating the use of constructive networks, this work argues that function approximators with a bounded memory (such as a fixed-architecture network) are less likely to overcome this systematic bias than those with an unbounded memory (such as a constructive network). [3] suggests the use of Cascor networks as a future development of TD-Gammon, to avoid the need to completely retrain the network when new features are added. More generally constructive algorithms are well-suited to tasks which require building on previously acquired knowledge. This would include various techniques previously discussed in the reinforcement learning literature such as transfer of knowledge between tasks [8], shaping [9] and improving on policies learnt through observation [10].

Within the localised approximator research, there has been a significant exploration of the use of constructive approximators which build their structure

during training, starting from a minimal architecture. For example [11] adapted the Resource Allocating Network to reinforcement learning, whilst more recently [12] explored the use of constructive Sparse Distributed Memories.

In contrast the use of constructive global approximators for reinforcement learning has been minimal, despite this style of system being widely and successfully applied within the supervised learning field. Many constructive algorithms have been proposed in the supervised learning literature, but amongst the most widely adopted has been Cascade-Correlation [13]. This constructive algorithm, based on non-localised neurons, has been shown to equal or outperform fixed-architecture networks on a wide range of supervised learning tasks [13, 14]. Despite this, the only previous work using a cascade constructive network for reinforcement learning appears to be that of [15, 16], which will be discussed further in Section 3.

2 Constructive Learning Algorithms

As mentioned above the constructive neural network algorithms explored in the supervised-learning literature can be divided into two categories depending on whether they use hidden neurons with local or global responsive activation functions. This section will describe on algorithm of each type, which have been used as the basis for the reinforcement-learning systems described in Sections 3 and 4.

2.1 Cascade Networks

A Cascade-Correlation (or Cascor) network starts with each input connected to every output neuron, with no hidden neurons. This network is trained to minimise the mean-squared-error on a set of training examples. The mean-squared error is monitored, and if it fails to fall sufficiently over recent training epochs (as determined by a patience threshold), the decision is made to add a new hidden neuron.

A pool of candidate neurons is created, with each receiving input from all input neurons, and from any prior hidden neurons. Each candidate is trained to maximise the correlation between its activation and the residual error on the training set. Once trained, the candidate with the highest correlation is added to the network. Its weights are frozen, and it is connected to the output neurons with new random weights. Training of the output weights is now resumed. This process of alternating training of output weights and of candidates continues until a suitable solution is found or a pre-defined maximum number of nodes have been added. It should be noted that the candidate neurons may use a variety of different activation functions, but most commonly sigmoidal functions are used.

Cascor produces a deeply layered network structure. The use of previous hidden neurons as input to new hidden neurons permits Cascor to form high-level feature detectors which can be beneficial on difficult problems. In addition the freezing of the hidden neurons' input weights means that only a single layer of weights is being trained at any point in time, which accelerates the training process.

2.2 Resource Allocating Networks

As with Cascor, a Resource Allocating Network (RAN) starts from a minimal architecture containing just input and output neurons. In the RAN architecture as originally proposed [17], the connection topology is even more minimal as the output neurons start with only a single bias weight. During training the RAN adds hidden units which take input from all the input neurons, and apply a gaussian activation function which produces a response which is localised to a small region of the state space.

Each time a new input pattern is presented to the network, it is evaluated against two novelty criteria. If the distance between this input pattern and the centre of the closest hidden neuron response function exceeds a distance threshold, and the network's error on this input pattern exceeds an error threshold, then a new hidden neuron is added to the network. The distance threshold is decayed over time, so that the network initially adds neurons with a broad response, with more finely-tuned neurons added later in training. The parameters for a new neuron are determined directly from the current input pattern. The centre of the activation function is set to the current input pattern, and the width of the activation function is set to a percentage of the distance to the centre of the nearest existing hidden neuron. Finally the weight of the connections from the new neuron to the output neurons is set to equal the current error at that output. In this way that error is immediately corrected.

If the current input does not meet the novelty criteria, then the weights of the output neurons and the centres of the hidden neurons are trained using standard gradient descent.

The RAN has two potential advantages over Cascor. First the direct initialisation of the hidden neuron parameters is potentially much faster than the training of the candidate neurons in Cascor. This is of particular benefit in an online learning context as it may allow more rapid improvements in performance during training. Second the localised response of the RAN's hidden neurons may be of benefit in problems involving discontinuities or rapid changes in the output function over the state space.

However the RAN's single hidden-layer architecture prevents it from forming the higher-level feature detectors which are available in a Cascade network. In addition the localised nature of the RAN hidden neuron response functions means that many more neurons may be required to solve a given problem. Therefore it is not clear *a priori* which of these architectures will be best suited to the task of value approximation within a reinforcement learning context.

3 Adapting Cascor to online reinforcement learning

There are a number of features of the Cascor algorithm which prevent it from being applied directly to the online learning of state or action values. The approach used in [15, 16] was to modify the reinforcement learning process so as to allow direct application of the Cascor algorithm. They propose a learning algorithm with two alternating stages. In the first stage the agent selects and executes actions, and stores the input state and the target value generated via TD in a cache. Once the cache is full,

a network is trained on the cached examples, using the standard Cascor algorithm. Once the network has been trained, the cache is cleared and the algorithm returns to the cache-filling phase. Results have been reported for tic-tac-toe, car-rental and backgammon tasks. The results for the first two tasks are promising, but the system did not perform well on the more complex backgammon task. In addition, the algorithm may not be suitable in real-time tasks due to the time requirements of the training phase.

In contrast we have made several modifications to the Cascor process to facilitate its direct incorporation into existing reinforcement learning algorithms. For the purposes of this paper this constructive training algorithm has been implemented and tested within the context of learning action values as part of the sarsa algorithm [18], but it should be equally applicable to other algorithms based on the method of temporal differences, such as Q-learning or learning of state values. As in [18] we use a separate single-output network for each action, rather than a single network with multiple outputs.

3.1 Choice of cascade algorithm

Several authors have demonstrated that Cascor, whilst effective for classification, is less successful on regression tasks. This is due to the correlation term tending to drive the hidden unit activations to their extreme values, thereby making it hard for the network to produce a smoothly varying output [19, 20]. The learning of state or action values is a regression task, and therefore Cascor may not be the most appropriate algorithm for this situation. [1] Several variants have been proposed to address this issue, based on the cascade architecture but using alternative training algorithms which do not include the correlation term [19, 21, 22].

In addition when used in supervised learning Cascor is usually implemented as a batch training algorithm. The complete set of examples in the training set are processed by the network prior to performing any weight updates, which facilitates the calculation of the correlation terms. In contrast the correlation values are less readily calculated in an on-line learning context where there is no fixed training set.

For these reasons we have used the Cascade2 (originally developed by Fahlman as reported in [21]) and Fixed Cascade Error [22] algorithms in preference to Cascor. Cascade2 differs from Cascor by training candidates to directly minimise the residual error rather than to maximise their correlation with that error. To facilitate this process, output weights are trained for each candidate in addition to its input weights, and these are also transplanted to the main network when a new hidden node is added. Fixed Cascade Error remains closer to the original Cascor strategy, but uses a different form of the objective function to be maximised which avoids the problems resulting from using the correlation term.

---

[1] Although, unlike most regression tasks, in this case the absolute accuracy of the function approximator is less relevant than its relative accuracy - as long as the optimal action for each state is valued higher than the other actions then the optimal policy will be followed. Therefore Cascor's accuracy issues may be less problematic in this context than for other regression tasks.

3.2 Choice of weight update algorithm

Whilst a range of weight update algorithms could be combined with the cascade training process, the original formulation and most implementations use the Quickprop algorithm [23]. Quickprop uses an estimate of the second derivative of the error with respect to the weights to produce faster training of the network. However Quickprop is designed for batch-training applications, and can not readily be adapted to the on-line training process where weights are updated after every forward pass of the network. Therefore for these experiments the backpropagation weight update algorithm has been used instead. The same training algorithm was also used for the multi-layer perceptron and RAN results reported here, so as to ensure any differences are due to the architectural aspects of the networks.

3.3 Serial or parallel training of candidates

In the supervised learning context, the cascade training process alternates between training the weights of the output neurons in the main network, and training the weights of the neurons in the candidate pool. The main network is trained until its performance ceases to improve, at which point its weights are temporarily frozen, and a new pool of candidates is produced and trained to reduce the residual error at the outputs of the main network.

This approach which we will refer to as 'serial candidate training' has several benefits. It facilitates the training of the candidate neurons by ensuring that the problem facing them is static, as the main network does not change whilst the candidates are training. It also allows the activations of the neurons in the main network on each of the examples in the training set to be cached prior to commencing candidate training, which provides significant savings in computation.

However serial candidate training has serious disadvantages in the context of on-line learning. During training of the output weights the agent interacts with the environment whilst simultaneously adjusting its behaviour. However during the candidate training phase the agent is still interacting with the environment, whilst the adjustments made to the candidate weights do not alter the agent's policy until a new hidden neuron is added at the conclusion of candidate training. Therefore the agent is executing actions within the environment for a considerable period of time whilst following a fixed, sub-optimal policy which will significantly impact on its on-line performance. The approach of [15, 16] partially addresses this issue by caching data to use in off-line training of the network. The time required to fill the cache will be much smaller than the time spent training the network and therefore this approach reduces the regret due to following a static, non-optimal policy, but it does not produce a truly on-line algorithm.

In addition the absence of a fixed set of training examples eliminates the possibility of reducing computation by caching the activations of the main network. The main network's activation will need to be recalculated for every state visited during the candidate training phase, as these states may not have been encountered previously.

For these reasons we have used parallel candidate training in this paper. After each interaction with the environment the temporal difference error δ_{TD} is calculated. The

eligibility traces and weights associated with the output neuron are then updated, as shown in equations 1 and 2, where w_i is the weight of the connection from the *i*th input/hidden neuron to the output neuron, e_i is the eligibility trace for that connection, I_i is the output of the *i*th input/hidden neuron, λ is the trace decay rate and α is the learning rate.

$$e_i' = \lambda e_i, \text{ if this action was not the most recently selected} \quad (1)$$

$$e_i' = I_i + \lambda e_i, \text{ if this action was the most recently selected}$$

$$\Delta w_i = \alpha \delta_{TD} e_i' \quad (2)$$

In parallel, training is performed for the candidate neurons. Eligibility trace and weight updates are performed for these candidates as if they were connected to the network, but they do not contribute to the activation of the output neuron. For the Cascade2 network, the input and output weights for the candidates are trained to reduce the residual error, by minimising the following term, where w_c is the output weight for the candidate neuron, and o_c is the current activation of the candidate neuron:

$$\delta_C = \delta_{TD} - w_c o_c \quad (3)$$

For the Fixed Cascade Error network, the candidate units are trained to maximise the value of the following function:

$$C_{FE} = \left(\delta_{TD} - \overline{\delta_{TD}}\right) o_c \quad (4)$$

A possible disadvantage of parallel candidate training, which is likely the reason it is not used in supervised learning, is the issue of moving targets. The values of δ_{TD} change during training as the weights of the output neurons are adapted. As these values form the targets for the candidate neuron training, the task facing the candidate neurons is more complex than it would be if these values were static as is the case in serial candidate training.

3.4 Patience testing across multiple networks

In Cascor, the output error is accumulated during training, and periodically tested against the patience threshold to decide whether to add a new node. In sarsa the same error (δ_{TD}) is used for all networks so basing the patience tests on this term would result in the same topology being grown for all networks. The function to be learnt may vary in complexity between the different actions, and so each network should be able to independently determine its own topology. One means for achieving this is to weight the δ_{TD} term based on its relevance to that particular network, which will vary depending on how recently the corresponding action has been selected. This is achieved by maintaining a replacing eligibility trace e_n for each network, and using the following cumulative weighted error term in the patience testing process (where P

is the number of episodes in the patience period, and t_E is the number of time-steps in episode E):[2]

$$\frac{\sum_{E=1}^{P}\sum_{t=1}^{t_E}\delta_{TD}{}^{2}e_n}{\sum_{E=1}^{P}\sum_{t=1}^{t_E}e_n} \quad (\mathbf{5})$$

As is evident from equation 5, we measure the patience period in terms of the number of episodes. For non-episodic tasks it would be necessary to measure the patience period in time-steps. This may have the additional benefit of aiding in creating suitable network topologies, as the networks corresponding to actions which are rarely selected will automatically have less opportunities to add a hidden neuron.

If the term in equation 5 fails to improve sufficiently on the value measured over the previous patience period then a candidate will be selected to add to the network. The performance metric used to select the best candidate is given in equation 6 (note that in this case there is no need to normalise this term by the sum of the eligibility traces as was done in equation 5, as this would not affect the relative ordering of the candidates). The candidate with the lowest value for this metric is selected to be added to the main network.

$$\sum_{E=1}^{P}\sum_{t=1}^{t_E}\delta_{C}{}^{2}e_n \quad (\mathbf{3})$$

As an implementation issue it should be noted that once a new hidden neuron is added, its input weights are frozen and so there is no need to maintain eligibility traces or perform weight updates for these weights.

3.5 The cascade-sarsa algorithm

Combining all of the issues discussed in Sections 3.1 to 3.4 yields the cascade-sarsa algorithm, as shown in Figure 1. The Cascade2 and Fixed Cascade Error variants differ only in the calculation of the eligibility traces.

---

[2] The approach given here differs in technique, though not intent, from that reported in an earlier paper on this work [24]. There we used the sum of the absolute values of the output weight traces rather than maintaining explicit action traces - whilst giving similar results on the problems tested, that approach is a potential source of bias as the weight eligibility traces depend on the value of the input and hidden neuron activations as well as the turns elapsed since this action was last selected. It is also worth noting that [24] contained an error in the off-line results reported for the Mountain-Car task due to incorrect sampling of the starting states - this has been rectified in the results reported in Section 5 of this paper.

```
P_last = +∞
while (!finished training) do
{
    P_current = 0
    for P learning episodes
    {
        clear all eligibility traces
        while (! end of the episode)
        {
            observe the current state of the environment
            calculate the output of each network
            select an action a e-greedily based on networks' outputs
            Q_t = output of network a
            if this action is not the first in the episode
                δ_TD = r + γQ_t - Q_t-1
                for each network
                    update output weights to minimise δ_TD
                    update candidate weights
                    recalculate network activations
            Q_t-1 = output of network a
            update eligibility traces for all weights in all networks
            execute action a and observe the reward r
            update Pcurrent as per equation 5
        }
    }
    if P_current > patience-threshold * Plast
        select best candidate (per equation 6) and add to the network
        P_last = +∞
    else
        P_last = P_current
}
```

Fig. 1. The Cascade-sarsa algorithm using one constructive neural network value approximator for each discrete action.

4 Adapting the RAN to online reinforcement learning

In contrast to Cascade-Correlation, the RAN algorithm was originally designed for on-line function approximation and therefore requires relatively little modification in order to adapt it for on-line value approximation. The major change is the requirement to maintain an eligibility trace for each of the output neuron weights.

As with the Cascade networks, a separate RAN was created for each possible action. As discussed in Section 3.4 it is preferable for each of these networks to be free to independently determine its own topology during training. Therefore we

maintain a replacing eligibility trace for each network, and weight the temporal difference error by this trace when testing an input pattern against the error threshold. It is important to note that this weighted error was only used for this purpose; all weight updates were calculated based on the original unweighted temporal difference error.

[5] previously reported that training the centres of existing hidden neurons via gradient-descent as in Platt's original supervised RAN was found to be detrimental to performance in a reinforcement learning context. Our experience agreed with this observation, and therefore in these experiments the centres of hidden neurons were fixed once added to the network, with only the output weights of the network subject to training via gradient-descent.

One other minor modification was made to the original RAN algorithm. Rather than starting with output neurons with connections only to a bias unit, we included shortcut connections from the inputs to the output neurons. These should facilitate more rapid learning of any linear aspects of the problem, and would be equally applicable in supervised learning using RANs.

5 Experimental method

5.1 Benchmark problems

The constructive reinforcement learning algorithms described in Sections 3 and 4 were compared against a multi-layer perceptron on three benchmark problems from the reinforcement learning literature - Acrobot, Mountain-Car and Puddleworld [4, 25]. These problems were chosen as [25] reports that a MLP was unable to learn a suitable policy on any of these tasks.

As illustrated in Figure 1(a) the Acrobot consists of a two-link robot, capable of applying torque only at the joint between the two links. The task is to raise the unconstrained tip of the second link to a height above the first joint equal to or greater than the length of a single link. The system is described by four continuous variables - the joints' angular positions and velocities. As the angular positions are cyclical in nature, each position is encoded by a pair of inputs storing the sine and cosine of the angle, to avoid discontinuities in the input as suggested by [26]. This results in a total of six input neurons for this task. There are three possible actions - exerting positive torque, negative torque or no torque. A penalty of -1 is received on all steps on which the goal-state is not reached.

As shown in Figure 1(b), the Mountain-Car task requires a car to escape from a 1-dimensional valley. The car’s engine is less powerful than gravity, and so the car must reverse up the left-hand side of the valley to build enough potential energy to escape from the right-hand side. The inputs to the agent are the car’s current position and velocity, and there are three possible actions - full throttle forward, full throttle backward, and zero throttle. As with Acrobot a penalty of -1 is received on all steps on which the goal-state is not reached.

As shown in Figure 1(c), Puddleworld is a two-dimensional environment. The agent starts each episode at a random, non-goal state and has to find its way to the goal. The agent receives its current coordinates as input, and at each step selects between four actions (left, right, up or down) which move it by 0.05 units in the desired direction. At each step a small amount of gaussian noise (standard deviation 0.01) is also added. The agent's position is bounded by the limits of the world (0...1). On each step on which the goal is not reached, the agent receives a penalty of -1. An additional penalty is applied when the agent is within a puddle, equal to 400 multiplied by the distance to the nearest edge of the puddle.

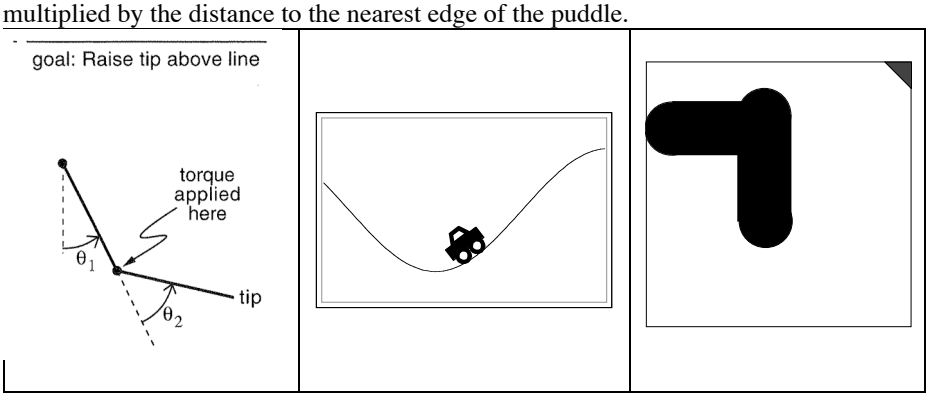

Fig. 2. (a) Acrobot (from [27, p. 271]) (b) Mountain-Car. The goal is to escape from the right-hand edge of the valley. (c) Puddleworld. The goal is the triangle in the top-right corner, and the puddles are capsules with radius 0.1, defined by the line segments (0.1, 0.75) to (0.45, 0.75), and (0.45, 0.4) to (0.45, 0.8).

5.2 Network types

Four styles of network were applied to each of the problems - a multi-layer perceptron with a single hidden layer of neurons using asymmetric sigmoid activation functions, Cascade2 and Fixed Cascade Error networks with candidates using the same asymmetric sigmoid function, and a RAN using radial basis activation functions. All networks used linear activation functions for their output neurons, for two reasons. The use of a non-linear activation function, whilst potentially useful in classifiers, can be detrimental on regression tasks. In addition, the use of a bounded function such as a sigmoid requires the actual values to be scaled to the range of this function. For the Puddleworld task in particular this is problematic, as the maximum negative reward theoretically possible is much higher than the largest value likely to actually be experienced. Using linear output neurons avoids the need to make any *a priori* decisions about scaling.

This decision lead to one further modification to the learning algorithms. The structure of the reinforcement signals requires the networks to produce large negative values. This means much larger weights are required on the connections to the output neurons than on the input connections of the hidden neurons. This leads to a problem of ill-conditioning similar to that described by [6] - the learning rate which is

appropriate for the output layer weights is not suitable for the hidden layer weights. In particular the suitable learning rate for the hidden layer weights falls over time as the output weights grow larger. To address this issue the calculation of the weight change for the hidden layer weights in the MLP was modified from that shown in equation X to that in equation X+1. Dividing by the absolute value of the output weight reduces the impact of the output weight magnitude on the rate of change in the hidden weights, whilst maintaining the correct direction for those changes.

A similar modification was made for the cascade network's weight updates. Early trials indicated that this normalisation process was extremely beneficial for the MLPs, and of lesser benefit to the cascade networks (as their output weights never grew to the same magnitude). This modification was not required for the RAN as the size of the output weights had no consequence on the weight updates, given that we were not training the centres of the radial basis functions.

4.3 Measuring performance

For each problem, a number of trials were run for each type of network to find appropriate values for the parameters (the learning rate α and eligibility trace decay factor λ for all types of network; the number of hidden nodes for the MLP; the patience period length for the cascade networks; and the error and distance thresholds for the RANs). A fixed patience threshold of 0.95 was used for all cascade network trials. For each set of parameters 20 networks were trained, with different initial random weights. Each network was trained over 1000 episodes, using ε-greedy selection. ε was set to 0.2 for the Puddleworld and Mountain-Car tasks, and to a lower value of 0.05 for the Acrobot task as non-optimal actions can have a greater detrimental effect in this task, as noted by [4]. Each episode ended either when the goal state was reached, or after a maximum number of time-steps (a limit of 1000 steps was used for the Mountain-Car and Puddleworld tasks, and 500 for the Acrobot trials). No discounting was used.

For each trial the learning system's on-line performance was assessed by calculating the mean reward received per episode. The parameter set yielding the highest mean reward was selected as the optimal settings for that style of network for that task. Whilst on-line performance is clearly an important measure of the performance of any on-line learning algorithm, this measure has the potential to be misleading as the choice of the number of episodes over which to measure performance introduces a source of bias. Algorithms which learn quickly but converge to a sub-optimal solution will be favoured by a shorter number of episodes, whilst algorithms which learn slowly but more accurately will be favoured by a longer assessment period. Therefore the quality of the final policy learnt by each network at the end of training was also measured. Two approaches were used. The first was the mean on-line reward received over the final 50 episodes of training - this will be referred to as the final on-line reward. The second approach was to measure the network's off-line performance. Following training, each network's policy was assessed by running a further set of episodes using strictly greedy selection, and with learning disabled.

5 Results and Discussion

Table 1 shows the results achieved by each of the network styles, for the best parameter set found for that type of network, averaged across all 20 networks trained using those parameters.

		MLP	Cascade 2	FCE	RAN
Acrobot	Parameters	$\lambda = .8$, $\alpha = 0.005$	P = 40, $\lambda = 1$, $\alpha = 0.001$	P = 60, $\lambda = 1$, $\alpha = 0.0005$	E = 2, $\lambda = 0.8$, $\alpha = 0.01$
	Hidden nodes	10.0	8.5	5.3	200.0
	On-line (all episodes)	-260.1	-122.0	-122.4	-120.1
	On-line (final)	-165.3	-108.0	-108.6	-111.0
	Off-line	-290.7	-103.4	-96.7	-102.4
Mountain-Car	Parameters	$\lambda = 0.8$, $\alpha = 0.001$	P = 60, $\lambda = 1$, $\alpha = 0.001$	P = 20, $\lambda = 1$, $\alpha = 0.001$	E = 2, $\lambda = 1$, $\alpha = 0.001$
	Hidden nodes	12.0	5.4	19.1	150.0
	On-line (all episodes)	-216.4	-91.5	-91.7	-87.6
	On-line (final)	-157.5	-83.0	-86.6	-74.1
	Off-line	-319.2	-77.4	-98.7	-91.4
Puddleworld	Parameters	$\lambda = .6$, $\alpha = 0.01$	P = 80, $\lambda = .9$, $\alpha = 0.001$	P = 40, $\lambda = 1$, $\alpha = 0.001$	E = 2, $\lambda = 0.7$, $\alpha = 0.001$
	Hidden nodes	12.0	4.7	4.0	129.7
	On-line (all episodes)	-163.9	-335.2	-435.8	-158.6
	On-line (final)	-103.5	-193.0	-255.5	-61.7
	Off-line	-406.9	-299.8	-301.6	-48.4

Table 1. Results on the Acrobot, Mountain-Car and Puddleworld tasks, for the best parameter set found for each style of network. The results are the means over 20 trials from different starting weights.

5.1 Comparison of the performance of the learning algorithms

In line with the findings of [25], the fixed-architecture network found these tasks difficult, faring poorly on all tasks in both on-line and particularly off-line performance. In contrast the RAN performed well on all three benchmarks for both the on-line and off-line measures. However this performance was only achieved by choosing parameters which allowed the RAN to create a very large number of hidden neurons. The Acrobot task which had the highest input dimensionality also resulted in

the largest network as every trial added the maximum allowed number of 200 neurons.

The performance of the two cascade constructive algorithms was less consistent. Both the Cascade2 and FCE networks were competitive with the RAN on the Acrobot and Mountain-Car tasks, and due to the use of non-locally-responsive neurons were able to achieve this level of performance using far fewer resources than required by the RAN. However both cascade algorithms failed spectacularly on the Puddleworld problem, failing to match even the relatively poor on-line results of the multi-layer perceptron.

Some insight into the reason for the inconsistent performance of the cascade algorithms is given by examining the policies learnt by a successful network on each of the tasks. The decision boundaries in the Mountain-Car task are primarily linear, and therefore each boundary can be learnt by a single sigmoidal hidden neuron. Hence the cascade algorithms perform well on these tasks as the candidate nodes can readily learn to improve the value approximation. In contrast the Puddleworld task requires the networks to form localised regions within the input space, which can not be achieved by a single hidden neuron. The general increase in cost from the top-right to bottom-left corner of input-space can readily be learnt by a cascade network with no hidden neurons. However any attempt at that stage to add a candidate neuron to correct the estimated values in the regions of input-space corresponding to the puddles will fail, as that neuron will adversely affect the estimates elsewhere in input-space due to the global nature of its activation function. In contrast the RAN's localised neurons can readily adjust the estimate in just the relevant regions of the input-space. Even the MLP is better suited to this problem than the cascade networks. As it trains multiple hidden neurons simultaneously, they can collectively produce the necessary localised change in the system's output, although clearly from the results this process is slower and less accurate than that of the RAN.

5.2 On-line versus off-line performance

Although not the major topic of this paper, one interesting aspect of the results is the variation between the final on-line reward and the off-line performance, particularly for the MLPs. As these are intended to be different measures of the same criteria it is on the surface surprising that the results are so different.

A closer inspection of the behaviour of some of the MLPs on the Puddleworld tasks provides some insight into this paradox. Examination of the final values indicates that the network has learnt a ranking of the actions which is independent of the state - the move left and down actions are always rated lower than the right and up actions, which have very similar values. During training episodes, the agent continually selects the same action (for example, moving right) until it reaches the edge of the world. Further movements in the same direction result in no movement, and so after a few turns the value of that action is trained downwards sufficiently that it is no longer the highest-valued action. At that point the network switches to the other high-valued action (moving up), and repeatedly executes that action until the goal is reached. During training this ‘dynamic’ policy is moderately successful - it is

guaranteed to reach the goal and provides some measure of puddle avoidance (as encountering a puddle will usually result in an immediate drop in the value of the action just selected). However in the off-line tests when weight updates are no longer being performed this policy will lead to the agent timing out on most episodes.

Similar behaviour has previously been observed by one of the authors in a reinforcement learning system using linear approximators to learn a robot learn-following task [28]. In that case the agent demonstrated a tendency during training to over-learn the direction of the most recent turn in the path - if the next turn was in the opposite direction then the robot would start to turn the wrong way only to correct its error after moving partially off the path. For that problem further training enabled the system to correctly learn a more stable policy which distinguished between actions on the basis of the state, whereas this does not appear to have occurred for many of the networks on the Puddleworld task.

6 Conclusion and Future Work

The general-purpose function approximation abilities of neural networks should make them a valuable tool for use in reinforcement learning, but previous research has found the performance of fixed-architecture networks to be unreliable when trained using temporal-difference methods. Previous work has suggested the use of constructive function approximators, with the vast majority of algorithms using approximators with locally-responsive units. This paper has presented two variations of a constructive algorithm using globally-responsive units, based on the cascade-correlation supervised-learning algorithm. It has been shown that for some problems these globally-constructive algorithms can provide similar performance to a locally-constructive algorithm (the Resource Allocating Network), whilst producing far more compact solutions. However the Puddleworld task exposed major failings in the globally-constructive algorithms as they struggle to deal with localised discontinuities in the value function.

The ability to generate compact solutions is a potentially valuable feature of globally constructive algorithms, as it offers the possibility of scaling more effectively to problems with high input dimensionality. However these benefits can only be realised if the weakness of these algorithms on problems such as the Puddleworld can be overcome. We intend to attack this problem from two directions. One approach is to modify the cascade algorithm to encourage the formation of groups of hidden neurons which together function as a locally-responsive unit. A second approach is to modify a locally-constructive algorithm such as the RAN to allow it to add less locally responsive units where they would prove useful. One means of accomplishing this may be to use neurons with a highly variable response function (such as the Adapative Response Function Neuron described in [29]), in conjunction with competitive learning.

References

1. Sutton, R.S. (1988). Learning to predict by the methods of temporal differences, Machine Learning, Vol. 3, pp 9-44.
2. Crites, R.H. and Barto, A.G. (1996), Improving Elevator Performance Using Reinforcement Learning, NIPS-8.
3. Tesauro, G. J. (1995), Temporal difference learning and TD-Gammon, Communications of the ACM. 38(3), pp.58-68.
4. Sutton R.S. (1996). Generalisation in reinforcement learning: Successful examples using sparse coarse coding. In Touretzky D.S., Mozer M.C., & Hasselmo M.E. (Eds.). Advances in Neural Information Processing Systems: Proceedings of the 1995 Conference (1038-1044). Cambridge, MA: The MIT Press.
5. Kretchmar, R. M. and C. W. Anderson (1997). Comparison of CMACs and RBFs for local function approximators in reinforcement learning. IEEE International Conference on Neural Networks.
6. Coulom, R. (2002). Feedforward Neural Networks in Reinforcement Learning Applied to High-dimensional Motor Control. ALT2002, Springer-Verlag.
7. Thrun, S. and Schwartz, A. (1993), Issues in Using Function Approximation for Reinforcement Learning, Proceedings of the Fourth Connectionist Models Summer School, Hillsdale, NJ, Dec 1993.
8. Perkins, T. J. and D. Precup (1999). Using Options for Knowledge Transfer in Reinforcement Learning, Technical Report 99-34, Department of Computer Science, University of Massachusetts.
9. Randlov, J. and P. Alstrom (1998), Learning to Drive a Bicycle Using Reinforcement Learning and Shaping, ICML-98
10. Nechyba, M. C. and J. A. Bagnell (1999), Stabilizing Human Control Strategies Through Reinforcement Learning, Proceedings of the IEEE Hong Kong Symp. on Robotics and Control
11. Anderson, C. W. (1993). Q-learning with hidden unit restarting. Advances in Neural Information Processing Systems.
12. Ratitch, B. and D. Precup (2004). Sparse Distributed Memories for On-Line Value-Based Reinforcement Learning. ECML.
13. Fahlman, S. E. and Lebiere, C. (1990). The Cascade-Correlation Learning Architecture. in Touretzky, D.S., Advances in Neural Information Processing II, Morgan Kauffman: 524-532.
14. Waugh, S.G. (1995), Extending and benchmarking Cascade-Correlation, PhD thesis, Department of Computer Science, University of Tasmania
15. Rivest, F. and D. Precup (2003). Combining TD-learning with Cascade-correlation Networks. Twentieth International Conference on Machine Learning, Washington DC.
16. Bellemare, M.G., Precup, D. and Rivest, F. (2004), Reinforcement Learning Using Cascade-Correlation Neural Networks, Technical Report RL-3.04, McGill University, Canada.
17. Platt, J. (1991). "A Resource-Allocating Network for Function Interpolation." Neural Computation **3**: 213-225.
18. Rummery, G. and M. Niranjan (1994). On-line Q-Learning Using Connectionist Systems. Cambridge, Cambridge University Engineering Department.
19. Adams, A. and S. Waugh (1995), Function Evaluation and the Cascade-Correlation Architecture, IEEE International Conference on Neural Networks. pp. 942-946.
20. Hwang, J.-H., S.-S. You, et al. (1996). "The Cascade-Correlation Learning: A Projection Pursuit Learning Perspective." IEEE Transactions on Neural Networks 7(2): 278-288.

21. Prechelt, L. (1997). Investigation of the CasCor Family of Learning Algorithms, in Neural Networks, 10 (5) : 885-896.
22. Lahnajarvi, J. J.T., Lehtokangas, M.I., Saarinen, J.P.P., (2002). Evaluation of constructive neural networks with cascaded architectures, in Neurocomputing 48: 573-607.
23. Fahlman, S. E. (1988) “Faster-Learning Variations on Back-Propagation: An Empirical Study”, Proceedings of the 1988 Connectionist Models Summer School
24. Vamplew, P and Ollington, R (2005), On-Line Reinforcement Learning Using Cascade Constructive Neural Networks, KES2005: Ninth International Conference on Knowledge-Based Intelligent Information & Engineering Systems
25. Boyan, J.A. and Moore, A.W. (1995), Generalization in reinforcement learning: Safely approximating the value function, NIPS-7.
26. Adams, A and Vamplew, P (1998), "Encoding and Decoding Cyclic Data", The South Pacific Journal of Natural Science, Vol 16, pp. 54-58
27. Sutton, R. and Barto, S. (1998), Reinforcement Learning, MIT Press
28. Vamplew, P. (2004), "Lego™ Mindstorms™ Robots as a Platform for Teaching Reinforcement Learning", at AISAT2004: International Conference on Artificial Intelligence in Science and Technology, Hobart, Australia, 21-25 November 2004.
29. Ollington, R. and Vamplew, P. (2003), Adaptive Response Function Neurons, 2nd International Conference on Computational Intelligence, Robotics and Autonomous Systems