Catastrophic Forgetting in Reinforcement-Learning Environments

Andrew Cahill

A thesis submitted for the degree of

Master of Science

at the University of Otago, Dunedin,

New Zealand

19 December 2010

Abstract

Reinforcement learning (RL) problems are a fundamental part of machine learning theory, and neural networks are one of the best known and most successful general tools for solving machine learning problems. Despite this, there is relatively little research concerning the combination of these two fundamental ideas. A few successful combined frameworks have been developed (Lin, 1992), but researchers often find that their implementations have unexpectedly poor performance (Rivest & Precup, 2003). One explanation for this is Catastrophic Forgetting (CF), a problem usually faced by neural networks when solving supervised sequential learning problems, made even more pressing in reinforcement learning. There are several techniques designed to alleviate the problem in supervised research, and this research investigates how useful they are in an RL context.

Previous researchers have comprehensively investigated Catastrophic Forgetting in many different types of supervised learning networks, and consequently this research focuses on the problem of CF in RL agents using neural networks for function approximation. There has been some previous research on CF in RL problems, but it has tended to be incomplete (Rivest & Precup, 2003), or involve complex many-layered, recurrent, constructive neural networks which can be difficult to understand and even more difficult to implement (Ring, 1994). Instead, this research aims to investigate CF in RL agents using simple feed-forward neural networks with a single hidden layer, and to apply the relatively simple approach of pseudorehearsal to solve reinforcement learning problems effectively. By doing so, we provide an easily implemented benchmark for more sophisticated continual learning RL agents, or a simple, 'good enough' continual learning agent that can avoid the problem of CF with reasonable efficiency. The open source RL-Glue framework was adopted for this research in an attempt to make the results more accessible to the RL research community (Tanner, 2008).

Acknowledgements

First of all, thanks to my supervisor Anthony Robins, for the endless support he provided throughout my postgraduate studies. Without the innumerable insights and sheer dedication he showed in reading and directing my research, this document could never have been completed. Many thanks also to my co-supervisor Nathan Rountree, who stepped in during Anthony's absence and ended up lending far more assistance than he bargained for!

Secondly, I am grateful to the University of Otago for the support they provided in the form of the very generous University of Otago Postgraduate Award.

Cheers to the entire Department of Computer Science and everyone who participated in the perennial afternoon teas, conferences, late night marking sessions, whiteboard tomfoolery and everything else, it has been a pleasure.

Last and most important is my fiancée Mary, my parents and my entire family. The support and encouragement you gave is more than I could ever ask for, and I will forever be grateful.

Glossary

CF	Catastrophic Forgetting
CHILD	Continual, Hierarchical, Incremental Learning and Development
DFR	Discounted Future Reward
DP	Dynamic Programming
FFBN	Feed-Forward Backpropagation (neural) Network
MDP	Markov Decision Problem
MLP	Multi-Layer Perceptron
NN	Neural Network
POMDP	Partially Observable Markov Decision Problem
RL	Reinforcement Learning
TD	Temporal Difference
TTH	Temporal Transition Hierarchies

Contents

List of Figures

1. Introduction

Humans, and nearly all other animals, are highly effective sequential learners. For example, a human can easily learn to tie his shoelaces, and then at some later date, to make spaghetti. If he is even slightly intelligent, he should be able to continue making use of these skills even after learning many more skills over the course of his life. This impressive talent is called sequential or lifelong learning ability, and is an active area of research in the artificial intelligence community.

Suppose you built a humanoid robot and endowed it with any widely used or well known machine learning system. A sensible way to do this would be to combine a neural network to give it memory and basic “reasoning”, and a temporal difference algorithm to enable it to learn motor control skills and explore its environment. Both neural networks (Rumelhart & McClelland, 1986) and learning by temporal difference (Schmidt, 2005) are analogous to processes in our own brains, and the algorithms are able to solve complex learning problems. Suppose you succeed in teaching it to tie its shoelaces and go on to teach it to make spaghetti. If you ask your robot to tie its shoelaces again, you will find it has completely forgotten how. In the field of neural network machine learning, this problem is called Catastrophic Forgetting, and it is as dramatic as the name suggests. In fact, even if after teaching it to tie its shoelaces you only attempt to teach it the tiny additional task of double knotting them so they stay tied for longer, you would have the same problem (to a slightly lesser degree).

Clearly, this must be resolved if we wish to build intelligent agents that can learn to solve a variety of real-world tasks. If our agent’s past knowledge is too easily disrupted then it cannot transfer its experiences to new problems. While a human child has the luxury of learning to walk before it learns to run, our agent has to continually struggle to solve difficult problems with little or no help from previous experiences.

On the other hand, there are already a few learning algorithms and systems that may be used to allow an agent to learn behaviours sequentially with minimal forgetting. Hierarchical reinforcement learning is one active avenue of research in this area (Russell & Norvig, 2003), and Temporal Transition Hierarchies (Ring, 1994), are one successful example of that type. There exist other potential solutions, such as Explanation Based Neural Networks, and Lifelong Learning Cell Structures. However, all of these algorithms are complex and some are only designed to solve certain specialised types of problems.

Fortunately, it may also be possible to learn and perform tasks in a sequential manner by simpler and more general methods, and even ones analogous to successful biological processes. Many researchers have investigated the simpler solutions that could be employed, but usually only in supervised function approximation problems (Robins, 2004; Robins, 1995; Ans, Rousset, Musca, & French, 2003; Sharkey & Sharkey, 1995). We will extend this research to problems with a different perspective, where we focus on the transfer of knowledge and lifelong learning ability of intelligent agents acting in virtual environments.

1.1 Machine learning

There are three major machine learning classes, largely distinguished by the level of feedback provided to the learning agent by its environment or teacher during the learning process. These are: unsupervised learning, reinforcement learning, and supervised learning, of which auto-associative learning/memorisation problems are a subclass.

In supervised learning problems there are, in general, a set of input patterns, a set of output patterns, and a teacher that knows the correct mappings between inputs and outputs. The agent in such a problem is tasked with learning a function that approximates the mapping that the teacher already knows. Learning proceeds as the agent is presented with an input pattern and predicts the associated output pattern. When the agent makes an error the teacher provides the correct output, and so the agent is able to tell exactly where it went wrong. Auto-associative problems are a subclass of supervised learning problems where the goal is for an agent to memorise a set of input patterns, for example to provide a content addressable memory (if the agent is able to generalise, it will return a correctly learned memory given a partial or corrupted pattern). Feedback in supervised problems tends to be much more complete and accurate relative to the other classes, meaning that learning is also more efficient.

Unsupervised problems lie at the other end of the feedback spectrum, and involve no feedback whatsoever from a teacher. They generally involve detecting statistical regularities in a dataset or population.

Reinforcement learning (RL) problems are the focus of this thesis, and they lie in the middle of the spectrum. As in supervised tasks, agents in reinforcement tasks are required to learn a mapping between inputs and outputs. Often the problem is stated differently, such that an agent is presented with inputs corresponding to observations of states in an environment (or similarly, readings provided by its sensors), and it is required to learn the optimal action to take given each input state. In contrast to supervised tasks, the RL agent is only provided with a ‘measure of goodness’ for its chosen action (output) instead of being told exactly what the optimal action was. Because there is less feedback, good solutions to RL problems are much harder to learn than in supervised problems. To make matters worse, feedback from the environment is often delayed for several steps, so the agent must learn to take actions that it predicts will eventually lead to the highest reward, instead of just choosing the action with the highest immediate return (a model called “greedy” action selection).

1.2 Project goals

This project aims to:

1. Investigate the problem of Catastrophic Forgetting (CF) as experienced by reinforcement learning agents with neural networks for function approximation.
2. Attempt to avoid CF in an RL agent by modifying a simple feed-forward neural network to use pseudorehearsal during learning.
3. Compare and contrast this result with other approaches, including basic rehearsal, context biasing, and Temporal Transition Hierarchies.
4. Use RL-glue as a platform to develop a simple neural-network based, function-approximating temporal-difference learning agent that avoids CF, for benchmarking against further RL and continual learning research.

1.3 Structure of this thesis

Chapter 2 is a brief introduction to the subject of neural networks and the multi-layer perceptrons used in this thesis. It also explains the problem of catastrophic forgetting in neural networks and the main classes of algorithms that cope better with the problem.

Chapter 3 explains the fundamental problem that this thesis is concerned with, and that autonomous intelligent agents need to be able to solve. It explains the various commonly used methods for solving reinforcement learning problems, and how agents using those methods model their environment and produce solutions. It also explains the more advanced complications to the basic problem and how the elementary solution methods can be adapted to it by using the algorithms presented in Chapter 2.

Chapter 4 is a description of the main investigation, including implementation of a set of sequential learning tasks, a reinforcement learning agent to solve them, and modifications that can improve its performance on these tasks. It details the experimental methods that were used and presents the results they generated. Chapter 5 is an analysis of these results and a discussion of their implications.

2. Neural Networks

This chapter introduces the neural network, a type of learning system that can be used to solve supervised learning problems, and that can generalise from things it has already learnt to novel situations. The usual definition of a neural network is an interconnected assembly of simple processing elements, units or nodes whose functionality is loosely based on the biological neuron. The processing ability of the network is stored in the inter-unit connection strengths, or weights, obtained by a process of adaptation to, or learning from, a set of training patterns (Gurney, 2007).

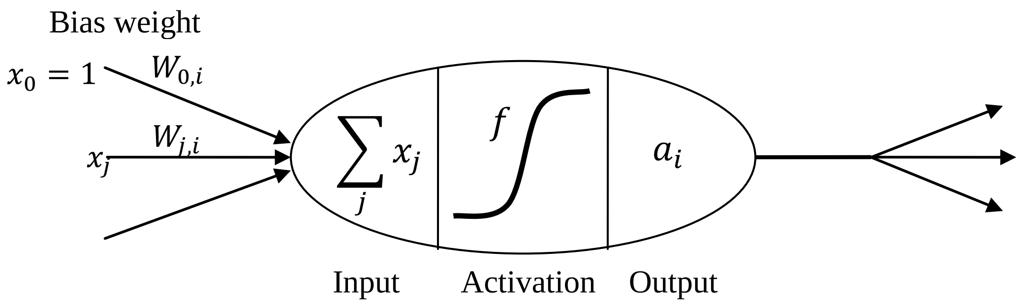


Figure 1: A neural network node i, following the model in Russell & Norvig (2003, p. 737)

The basic component of a neural network is a unit such as the one depicted in Figure 1. Each unit is linked to other units by directed connections which may either be inputs to or outputs from the unit. Additionally, each connection has an associated weight W describing the strength of the connection. The unit computes a weighted sum in of its inputs in the form:

$$in = \sum_{j=0}^{j} W_{j,i} x_j \qquad \text{Eqn. (1)}$$

It then applies an activation function f to in to derive the unit's output $a = f(in)$. The activation function for the unit in Figure 1 is a sigmoid function, so that for most values of the weighted sum the unit will be either inactive (output approximately 0), or active (approximately 1). However, there are many other possible activation functions, and a step function or sigmoid function (as in Figure 1) is often used for this task. A bias weight W_0 is also often used to influence the threshold of the activation function.

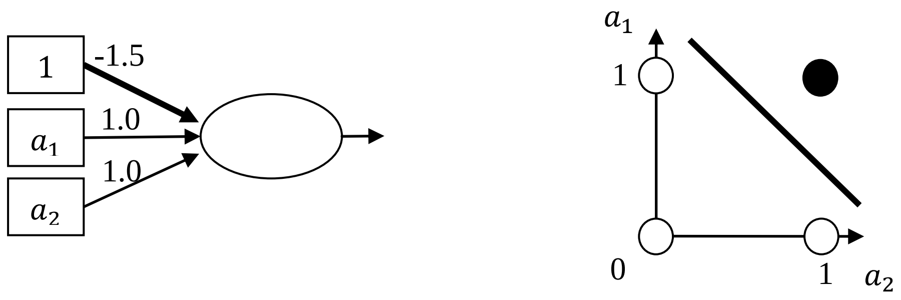


Figure 2: A neural network solving $a_1 \land a_2$, and a graph of the decision surface it creates

One of these units is sufficient to represent simple functions, or perform simple classification tasks. For example, we can represent the Boolean AND function by setting the weights of a unit with a step function for activation and two inputs as shown in Figure 2**.** The right hand side of the figure is a chart of the "space" of possible inputs (the four possible combinations of the binary inputs) and the desired output for each (1 (black) for [1, 1], and 0 (white) otherwise). The line represents the decision boundary created by the unit. In general, the input weights set the orientation of the surface and the bias weight sets the surface's position relative to the origin. In this case the unit correctly classifies every combination of inputs as a Boolean AND function would, so we can say it approximates (perfectly, in this case) the function. We also could use an array of units to compute a series of functions on an input pattern to test whether specific features are present in an input pattern.

However, a single unit or layer of units suffers from the key limitation that it can only solve linearly separable tasks. This means that it must be possible to place a single dividing surface in the input space that correctly separates the inputs according the classification we want the network to make (Russell & Norvig, 2003). In two dimensions this surface is simply a line, while in N dimensions it is a hyperplane. This division is possible for the Boolean AND function as shown, but not for many other functions. The simplest example of the problem is Boolean XOR, shown in Figure 3. In that case there is no way to divide the input space correctly with a single line, so the unit (or any number of units in a single layer) is unable to classify the inputs correctly. Fortunately, we can work around this by building a slightly more complicated network of units, also shown in Figure 3.

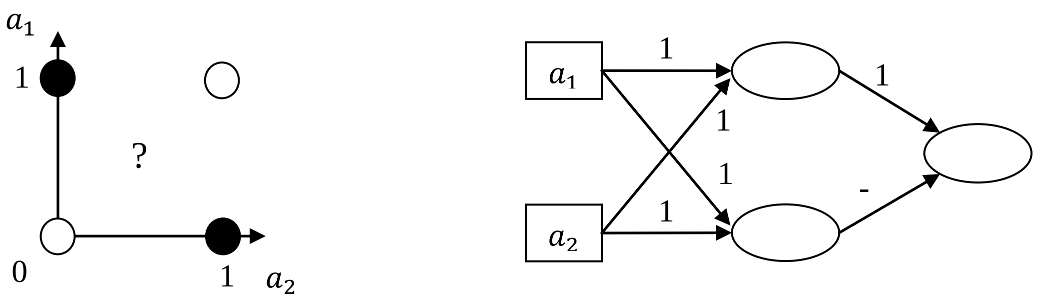


Figure 3: Boolean XOR and a neural network that solves it (bias weights noted inside units)

There are a vast number of possible architectures for a neural network, and no attempt will be made to explain them all here. The main investigation of this thesis starts with what is probably the most basic and standard of all NNs: a deterministic, fully-connected, feedforward network with one hidden layer, using the standard generalised delta rule combined with backpropagation[1] for supervised learning (Rumelhart & McClelland, 1986). This type of neural network is also called a Multi-Layer Perceptron (MLP).

2.1 Multi-layer perceptron learning

The example in Figure 3 is a simple Multi-Layer Perceptron with a solution to the XOR problem encoded in its connection weights. But how can we find the solution if we do not already know what the values of the weights should be? Normally when we create the network we initialise each of the weights to a small random value[2], and then train them to the final result.

The basic supervised learning approach to training is to set the input layer activations of the MLP to match a pattern we want to learn, and then propagate the inputs forward until the output layer is reached. The activation of each unit in each layer must be calculated in turn using the weighted sum formula in Equation 1 and the activation function for the node. At the end of this process we can compare the output layer to the final result we want the network to output, and alter the weights of the network to be closer to that ideal.

This is not quite as simple as it sounds however, because if an output node has an incorrect activation it is not immediately apparent if the error was caused by poorly trained weights on that node, or whether the error was in the activation of one of the hidden units supplying that

[1] Hereafter, this thesis adopts the common convention of referring to the combination of the generalised delta rule and backpropagation as the "backpropagation algorithm"

[2] The reason for starting with the network with random weights instead of just zero values is to prevent the problem of "symmetry locking" (Rumelhart & McClelland, 1986).

node. This is called the *structural credit assignment problem* (Gurney, 2007, p. 343). There are a variety of algorithms that decide how to correct the errors in network output, but the most fundamental and widely applicable is *backpropagation.*

2.2 Backpropagation

The basic idea of this algorithm is to reduce the error in each layer of the network in turn, starting with the output layer. The error in a single unit i in the output layer is $\Delta i = f'(in_i) \times (y_i - a_i)$; that is, the derived activation function applied to the unit's input, multiplied by the difference between the actual and desired output for that node. This error is used to update every input weight connecting a node j in the hidden layer to the node i, using the rule $W_{j,i} \leftarrow W_{j,i} + \mathrm{A} \times (l-1)_j \times \Delta i$. The symbol A is a learning rate parameter that controls how large the changes to the weights are in proportion to the error, and is usually kept small to avoid overshooting and oscillating around the correct value. The component $(l-1)_j$ refers to the activation of the unit in the previous layer connected to i by the weight $W_{j,i}$. After the output layer's error has been corrected we backpropagate the error through each hidden layer, by calculating the error Δj for each node in the layer l, which represents what fraction of the error that node is "responsible" for. The formula for Δj is $\Delta j = f'(in_j) \sum_i W_{j,i} \Delta i$. We can then update each unit's weights using the same update rule, and repeat the process for each layer until the input layer is reached (Russell & Norvig, 2003, p. 746).

To train the network completely we simply carry out this update process on each example in our training set in turn, and repeat it until the total error for all of the training examples is acceptably small. The total error of the network is the sum of the square of each individual error in the output (to avoid errors of different signs cancelling each other out), which we can use to calculate the error over the entire population of input-output examples. This is something of an oversimplification as in reality determining when to stop training can actually be quite difficult. Over-training affects the capability of the network to generalise and under-training allows unnecessary error. Strategies for choosing the right amount of training in a systematic way are outlined in Deco & Obradovic (1996, p. 31).

2.3 Continual learning and catastrophic forgetting

The MLP and learning algorithm described above fits readily into the usual model of supervised learning, where we have a representative set of input and output examples that we want to use to approximate a function or model a concept. However, as was briefly noted in the introductory sections of this thesis, neural networks perform very poorly when used for continual (also called serial or sequential) learning (Sharkey & Sharkey, 1995). Normal usage of the backpropagation algorithm is to have all of the data available at once and presented in sequence repeatedly, a situation known as *concurrent learning* (Eastman, 2005). However, as Eastman also explains, there may be times when this ideal situation is not possible. This could be the case if the training set is too large to conveniently fit in memory for the duration of training. Another example is if the training data is growing or changing over time, so that ongoing learning is required to maintain performance as the current version of dataset changes.

The problem is that a standard MLP using backpropagation for learning has excessive *plasticity*, meaning that it changes too quickly and too easily (Robins, 1995). Changes to incorporate new data, even small amounts of it, tend to affect the entire network and quickly disrupt the patterns already stored in the network weights. Figure 4 is an example of the process of catastrophic forgetting, adapted from Frean & Robins (1998). The network used to learn the training data had only a single input and a single output value (so the function it modelled could be easily plotted), but 20 hidden units. Each point on the graphs represents a mapping between an input (0-1) and an output (0-1), while the curve shows the output that the network gives for input values not explicitly taught to the network. Figure 4 (a) shows the network trained on six data points. Figure 4 (b) Shows the state of the network after learning a single new data point. Note that the network returns the correct value for the new data point, but all of the old data points have been disrupted. Figure 4 (c) shows the ideal state of the network, where catastrophic forgetting has not occurred and the correct value is returned at every point. In this case, the final figure is the result of training using a pseudorehearsal scheme, explained in Section 2.4.2.

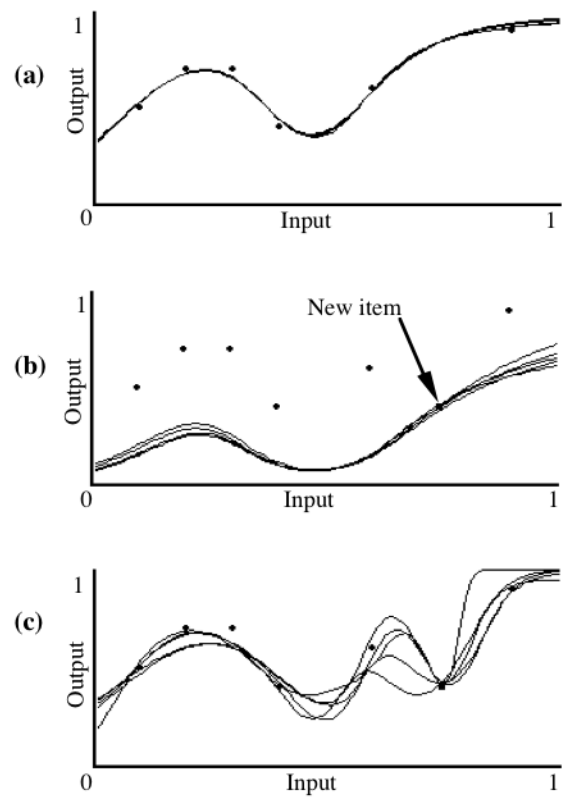


Figure 4: Functions learned by a network (five replications) adapted from Frean & Robins (1998)

Fixed-architecture feedforward MLPs are not the only type of neural network that suffers from the problem of catastrophic forgetting. In fact, it seems to be a general property of nearly every type of supervised network learning algorithm that was not specifically designed or modified to avoid it. It was noted by Eastman (2005) that constructive networks (specifically cascade correlation networks, but quite likely others as well) have the same problem despite predictions to the contrary, and Robins & McCallum (1998) discovered that Hopfield networks are equally susceptible. Similarly, recurrent (Elman) networks have the same weakness (Ans , Rousset, French, & Musca, 2002). Consequently, there has been a great deal of research into mitigating or avoiding its effects.

2.4 Methods for avoiding CF in MLPs

We already know that MLPs are good at concurrently learning a training set by backpropagation. It should be obvious then, that the simplest way to make a network retain its existing knowledge when it is required to learn a new mapping is to just add the mapping to the set of mappings it has already learned, and concurrently retrain the network on the entire set. This would be called *full* rehearsal, and would fit the new pattern into the function the network has learned without damaging recall of the existing mappings. However, this approach is not very efficient and it would be foolish to retrain the network completely if we only wanted to add one mapping to a set of thousands.

2.4.1 Rehearsal-based methods

There are several schemes that attempt to preserve the shape of the function the network approximates without preserving every single mapping the network has learned, by adding a new mapping to a set of existing mappings that are *representative* of the function contained in the network. These schemes are described more completely in Robins (1995). The best of these schemes was *sweep rehearsal*, where patterns are all stored separate to the network as they are encountered, but only a small number are used in a dynamic training buffer while learning new items. The buffer is populated by randomly selecting a number (Robins used a 3:1 rehearsal to novel item ratio) of trained patterns from the store, and is repopulated each *epoch* (a single presentation of the new item and every item in the rehearsal buffer). This scheme gives a “broad but shallow” sort of rehearsal, since more of the stored items are presented but none of them are trained completely (not counting the new item being trained). This system displays an impressive ability to maintain an existing (base) population of mappings while learning new items, but it has a number of drawbacks:

- It is necessary to retain all of the items learned by the network, which might not be feasible (for large populations), and arguably makes the network redundant as a memory store. Since neural networks are usually employed for their powers of generalisation and not their data-storing abilities this is not necessarily an issue, but it is worthy of consideration.
- Additionally, in situations where there is significant conceptual drift (target mappings change over time), there is a serious question: what do we do when the target for a mapping changes? We could store a new version of the pattern with the new target, but then we could wind up with an infinite number of slightly different mappings in

the store. We could also discard the old target, discard the new target, or maintain some sort of running average of the target. Which is used would likely depend on the specific problem being solved.

- Since we have already trained the network to perfection on the original population, training with them again in the rehearsal stage introduces the serious risk of *overtraining* the network on the base population. This could damage the network's ability to generalise to new data (Deco & Obradovic, 1996).

2.4.2 Pseudorehearsal

Robins suggests an alternative scheme dubbed *pseudorehearsal* that overcomes the above mentioned drawbacks of sweep rehearsal. In this case we carry out the same training regimen as in sweep rehearsal, but instead of populating the rehearsal buffer with stored items it should be filled with *pseudoitems* (hence the name). Pseudoitems are purely abstract mappings generated by feeding a random input into the network (for example, random ones and zeroes into a binary network) and using whatever the network outputs as the target for the input. The inputs used to generate the items do not have to be taken from examples that the network has already been presented, and they do not even have to be valid or sensible inputs for the problem at hand (for example, where inputs map to sensor readings or game or FSM states). However, Baddeley (2008) suggested that making an effort to approximate the input distribution may improve learning performance in at least some cases.

The performance of this scheme is marginally worse at maintaining a base population since we are only using the network's approximation and not the actual base population. However, this may actually turn out to be beneficial. As long as we use pseudoitems we are only rehearsing knowledge the network already has, so there can be no risk of overtraining the network on actual items[3]. Additionally, we no longer need to keep a store of rehearsal items because we can simply produce as many pseudoitems as we wish to use whenever we need them. Finally, this scheme conveniently deals with the problem of conceptual drift by eliminating any possibility of choice on the part of the implementer; targets are simply whatever the network has converged on.

A considerable amount of work has gone into designing neural networks that avoid catastrophic forgetting by employing pseudopatterns internally, instead of the standard

[3] Simply training a network on a set of pseudoitems containing no new mappings would be equivalent to applying the identity function to the network's weight matrix.

scheme explained above, where the pseudopattern rehearsal is an external process acting on the network (Robins, 2004, p. 19 has a review of these algorithms). The goal of this research is to make it easier to compare this strategy for continual learning to structures and processes residing within human brains, to better establish how continual learning works in actual brains. The most recent new system in the area was presented in Ans (2004). This system uses a more biologically realistic learning rule (compared to backpropagation) and has a "self-refreshing memory", where clusters of units and recurrent weights are used to generate pseudopatterns and train the forward weights with them during learning to avoid interference. As Robins (2004) states, these systems generally exhibit worse performance than the basic scheme, in part because they are designed less for performance and more for biological plausibility and other measures. They are also equivalent in concept to the basic scheme, so there is no real benefit in reproducing them in this research.

2.4.3 Transfer of knowledge

Another type of lifelong learning strategy is based on the idea of improving learning by transferring knowledge between related tasks. Explanation based neural network learning (Thrun & Mitchell, 1995) is a learning algorithm that works with a feedforward MLP, designed to quickly learn a series of robotics navigation and control tasks (inside a Q-learning agent system[4]) by drawing on their earlier experience solving related tasks. The idea is similar to a rehearsal scheme, except that "support sets" are used to speed up learning and increase generalisation on new data, instead of a rehearsal buffer being used to prevent disruption of old data. Support sets are a collection of previous training examples that are deliberately similar to the current task being faced. By using similar training examples, new examples tend to adopt similar representations to the support sets, so learning is a matter of fine-tuning and generalisation is more robust. Most research in this area does not concern itself directly with the problem of catastrophic forgetting, but EBNNs are apparently successful at serial learning tasks.

[4] Section 3.4 has an explanation of the details of Q-learning.

2.4.4 Methods of avoiding forgetting in standard neural networks

Backpropagation is a global learning algorithm, meaning that updates can easily affect the entire network. Consequently, different mappings learned by a network during normal concurrent training will tend to have considerable overlap in the way they are represented in the network. Changing or introducing any single pattern will tend to disrupt all of the others to some degree, depending on the level of overlap and the magnitude of the change. Several authors (French, 1999; Kanerva, 1988; Kruschke, 1991) have demonstrated that serial learning performance can be improved by reducing the overlap between ("orthogonalising") the representations of patterns in a neural network.

French (1991) suggested measuring this representational overlap as the degree of overlap in the activations of the hidden-layer nodes created by each input pattern. The interesting thing about this is that the actual information learned by a neural network is stored in the network's weights; the hidden layer activations are a highly transient phenomenon, representing a partial working that the network generates as a side-effect of retrieving information. French suggests that the hidden layer activations indicate which weights are most relevant to the pattern (since if a hidden layer node has no activation then it does not matter what value the weights connected to it have) and so this indicates which region of the weight space is used in the representation of that pattern.

French (1991) also developed the technique of activation sharpening, a good example of a simple strategy for reducing overlap in network representations. The goal of activation sharpening is to reliably produce semi-distributed representations that are local enough to overcome catastrophic forgetting yet are sufficiently distributed to permit generalization. Essentially, this is done by slightly increasing the activation of the most active hidden units, slightly decreasing the activation of the other units, and then changing the input-to-hidden layer weights to accommodate the modifications. The activation adustments use the formula:

$$\begin{array}{lll} A_{new} \leftarrow A_{old} + \alpha(1 - A_{old}) & \text{for the nodes to be sharpened;} & \\ A_{new} \leftarrow A_{old} - \alpha A_{old} & \text{for all other nodes.} & \end{array} \qquad \text{Eqn. (2)}$$

α is the sharpening factor. The weight adjustment is calculated by taking the difference between the old and new activations as the error for each node and performing standard backpropagation from the hidden to the input layer. After this extra initial step the usual execution and backpropagation procedure is performed for the pattern.

2.4.4.1 Context biasing

However, as French (1994) notes, restricting the representation space of the hidden layer also tends to reduce the capactiy of the network (so that a larger network is needed for the same task, resulting in decreased performance) and in some cases may actually reduce sequential learning performance. He attempted to improve on this by forcing hidden layer representations to be both well separated (orthogonal) and also distributed across the hidden layer, with a system called context-biasing.

Context biasing is the same as activation sharpening except that the Hamming distance[5] between the new target (the "teacher", in supervised learning) and the previous one is used in the activation modification rule. The new representation is separated from the previous one by modifying the hidden layer activation (A) of each node of the new representation according to the rule:

$$\begin{aligned} &if\ A_{new} \geq\ A_{previous}\ then\ A_{new} \leftarrow A_{new} + \alpha\beta(1 - A_{new}) \\ &else\ if\ A_{new} <\ A_{previous}\ then\ A_{new} \leftarrow (1 - \alpha\beta)A_{new} \end{aligned} \qquad \text{Eqn. (3)}$$

Where α is the Hamming distance and β is a biasing coefficient**.** Instead of continually retraining the output of the network to prevent it being disrupted, these systems aim to improve serial learning performance by changing the way knowledge is represented in the network. Robins (2004) notes that reduced overlap methods do not actually prevent CF when measured by the ability of the network to correctly reproduce previously learned outputs. Instead, they mitigate its effects to the extent that after new learning, old information can be more quickly relearned by the network (a "savings" measure). However, French found that relearning time was reduced by 50% with this method, and initial training time was decreased as well.

2.4.4.2 Lifelong learning Cell Structures

Another similar idea, introduced by Hamker (2001), is lifelong-learning Cell Structures, which he describes as a constructive Radial Basis Function-like system that has the same orthogonalising properties combined with the ability to add units to allow "lifelong" learning as with Temporal Transition Hierarchies (cf. Section 2.4.5). Hamker chose a Radial Basis Function system because the learning algorithm is relatively local compared to the global backpropagation algorithm, and so interference may be reduced compared to backpropagation.

[5] The Hamming distance between two binary strings is the number of bits that must be changed to transform one string into the other.

2.4.5 Temporal Transition Hierarchies

Another neural-network based system for serial learning is Temporal Transition Hierarchies (TTH), introduced in Ring (1994). The TTH algorithm uses neither a standard nonlinear MLP nor backpropagation, but it is highly efficient at solving Partially Observable Markov Decision Problems (refer to Section 3.7 for details) when used with a standard Q-learning algorithm (Section 3.8).

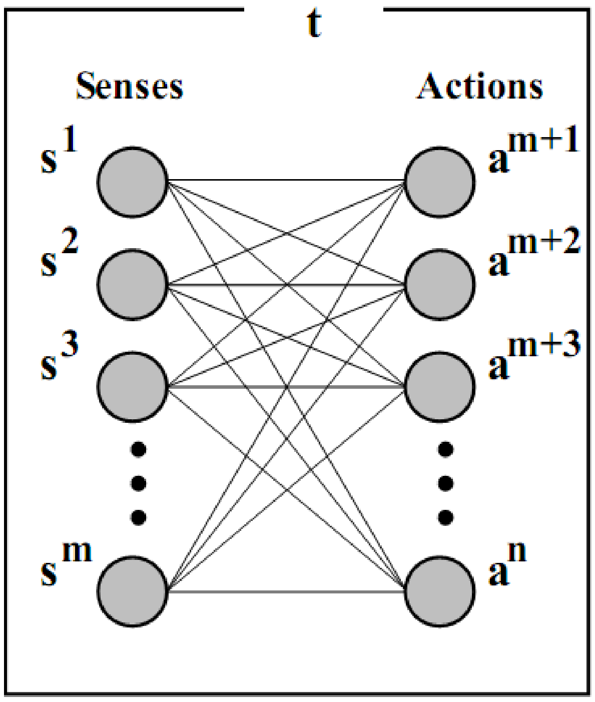


Figure 5: The initial structure of a Temporal Transition Hierarchies network

The initial structure of a TTH network (Figure 5) is a fully connected network (with no hidden layer) comprised of units with linear activation functions. The connection weights are at first improved as much as possible by the standard delta learning rule. When a connection weight is unable to converge adequately (determined by statistics that keep track of how often a weight is changed), a hidden unit is added with the sole purpose of reducing the error of that weight. The hidden unit is also connected to all of the agent senses. However, instead of directly propagating its activation forward as in an MLP, its activation modifies the connection weight it is assigned to, positively or negatively. This modification takes effect in the next time step, effectively causing the normal input to be amplified or reduced based on what the higher level unit sensed in the previous time step.

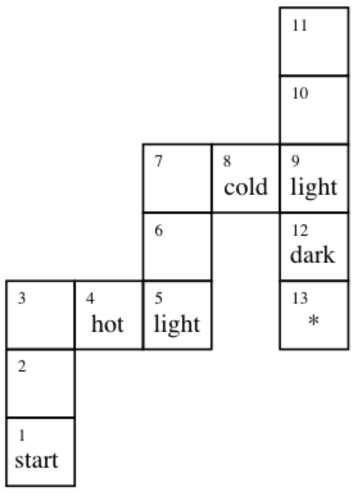


Figure 6: A simple maze test, where the agent has only four senses, and no positional knowledge

Figure 6 shows a simple grid-maze problem of the type which TTH is well-suited to solving. The task is to find the goal (13) from the start position (1) (or any random position in the maze) by choosing to move north, south, east or west. The agent moving around the maze has sensors for hot, cold, light and dark and receives their input before choosing the next step, but no other information is provided to it. The problem is a temporal one, because if the agent senses light it can only know whether to move north or south if it remembers what it sensed in the previous step.

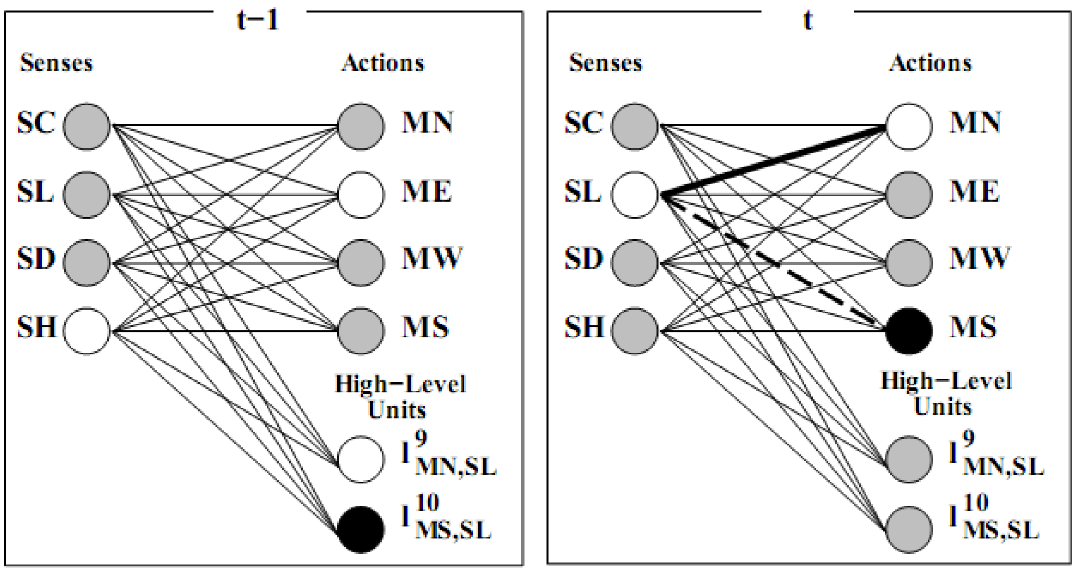


Figure 7: A TTH network trained to solve a maze task. Black, grey, and white represent -1, 0, and 1, respectively

Figure 7 shows a TTH network that can find the goal from every place in the maze (connection weights excluded). It has inputs SC, SL, SD and SH to sense cold, light, dark, and heat, and outputs MN, ME, MW, MS that serve as commands to move in each compass direction. It also has two high-level units that allow the network to make the context-sensitive decisions necessary to solve the problem. For example, at position 4 in the maze at time t-1 unit 9 (MN, SL) is positively activated because heat was sensed, and unit 10 (MS, SL) is negatively activated for the same reason. These activations persist until the next step t, where they increase that agent's tendency to choose north and decrease its tendency to choose to move south. The solid black line in Figure 7 (RHS) shows the positively modified connection, while the dotted line shows the negatively activated connection.

High-level units can also be assigned to modify the connection weights of other (lower level) high level units. These units' activations reach further back into the past the further away they are from the initial units (action units), allowing the network to incorporate knowledge from the arbitrarily distant past and therefore solve arbitrarily complex temporal problems.

2.5 Summary

This chapter introduced the class of supervised machine learning problems (in the context of neural networks) and techniques for solving them, including some of the basic supervised neural network algorithms. We also introduced the sub-problem of continual supervised learning, and catastrophic forgetting, a particular limitation of neural networks which is exposed by the continual learning problems. Several methods for overcoming catastrophic forgetting were discussed, including rehearsal, pseudorehearsal, context biasing, Temporal Transition Hierarchies, and others.

3. Reinforcement learning

Chapter Two gave an introduction to the class of supervised machine learning problems, which were first mentioned in Section 1.1. Supervised learning is probably the most often studied problem in the field of ML research, and most learning systems are developed to solve that type of problem. However, it is relatively rare to find tasks in the real world that can be easily framed as a supervised learning problem, simply because of the high level of feedback involved. For most problems, if the agent solving the task makes a mistake or sub-optimal decision there is no all-knowing teacher to describe exactly what should have been done instead, but this is exactly what is required to use supervised learning techniques (such as backpropagation) directly. More often, a teacher will be only able to provide limited feedback, possibly only after the task has been completed.

For example, when a (literal) teacher evaluates a student's paper they give it a grade and maybe attach some comments, but they would normally stop short of rewriting the entire paper, suggesting exactly which word should have gone where. They certainly would not stand over the student while he writes his paper and correct him each time he writes the wrong word, although we would need this perfect feedback for a supervised agent.

Alternatively, feedback can come directly from the environment. For example, an animal will quickly learn not to injure itself because when it is hurt it feels pain, a negative feedback signal resulting from its actions. However, the environment is unable to tell the animal how exactly that pain could have been avoided; the animal needs to work it out on its own. In studying reinforcement learning problems it is convenient to treat the 'agent' as just the brain or controller of the larger system, robot or creature, with any wheels, arms or anything else treated as part of the environment. Then we only need to concern ourselves with developing a mapping between inputs and outputs (which are connected to sensors and actuators) that allow the agent to perform some desired task.

The ability to learn to perform a simple task and then use that knowledge to solve a related but slightly harder task is especially important for learning agents. This incremental method is commonly called *shaping* and is a type of reinforcement learning. For example, the usual way to train a pigeon in a cage to push a button is to start by rewarding it whenever it crosses into the half of the cage where the button is located. When it has learned to stay in that half of the cage, the reward area can be reduced until the pigeon stays near the button, and then only when its beak moves near the button. Eventually, even when placed in an unfamiliar cage, the bird should make a beeline for any buttons in sight (Ring, 1994, p. 103).

3.1 Formal problems in RL

The examples above are informal and not precisely stated, but in fact the theory of RL sits on a solid mathematical foundation, and the most important problems are more exactly specified. The simplest and most fundamental formally stated problem in RL is known as the "n-armed bandit problem", named after the "one armed bandit" slot machines of casinos. Sutton & Barto (2005) explains it simply: Imagine you are repeatedly faced with a choice among *n* different options, or actions. After each choice you receive a numerical reward chosen from a stationary probability distribution that depends on the action you selected. Your objective is to maximize the expected total reward over some time period, for example, over 1000 action selections. Each action selection is called a *play*. How do you go about maximising your return?

This simple problem turns out to contain a great deal of underlying complexity, and it generalises to the more sophisticated problems we are interested in solving in this research. The defining feature of this type of problem is that there is no-one to tell you what lever you should have pulled at each play; instead, the environment functions as a teacher giving limited feedback. In addition, you have to choose between exploiting actions (lever selections) that appear to give good returns and trying new things to see if they work better.

3.2 Markov decision problems

Consider an RL agent trying to learn the task of behaving in the Wumpus World, pictured in Figure 8:

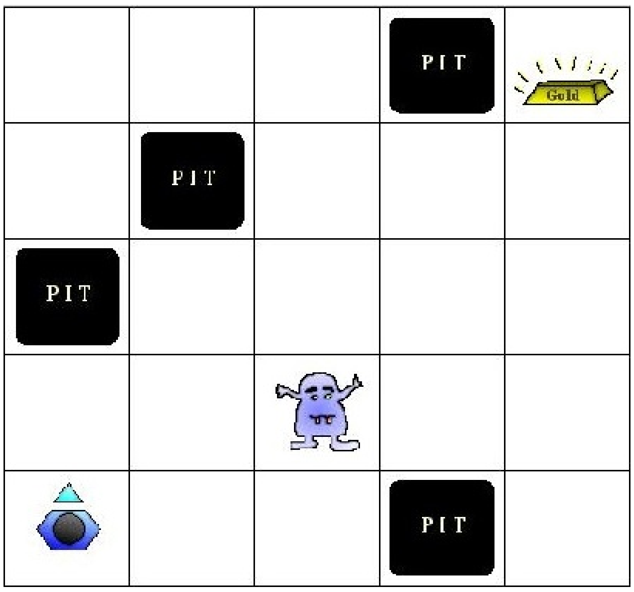


Figure 8: The Wumpus World

In this simple environment there are several pits, a monster, a reward/goal, and the agent (starting in the bottom left corner). If the agent blunders into the monster it receives a very large negative feedback (or punishment, for being eaten), a small negative feedback for falling in a pit (for being delayed/hurt), or a large positive reward for finding the goal. However, in the unmarked tiles the agent receives no feedback at all! To perform well in this environment, the agent would need to learn to take actions that lead to the goal, and avoid moving anywhere dangerous.

The Wumpus World is an example of a sequential decision problem, named as such because the agent needs to make a sequence of decisions to obtain a result and earlier decisions can have an effect on later decisions. Sequential decision problems are defined by three components: an initial state, a reward function (the reward associated with each state), and a transition model (the probability of reaching a state s' if action a is taken from a state s). The transition models used in this research are deterministic, so that if the agent takes an action a in a state s it will always reach the same s', but this need not be the case in general.

One important point about this environment is that it can be viewed as a collection of n-armed bandit problems, where each state is an n-armed bandit and each action is one of the arms. The difference is that each play takes you to a whole new bandit with a completely new probability distribution. Learning to behave in this sort of environment is very difficult, even with the simplified probability distribution of the Wumpus World.

The problems this research is concerned with are called Markov Decision Problems (MDPs), which are sequential decision problems with the added restriction of a Markovian transition model. This simply means that the probability of reaching state s' from s depends only on s and not on any earlier states. Study of MDPs makes up the majority of research into fundamental reinforcement learning methods (Sutton & Barto, 2005), and accordingly, they are the primary focus of this research.

There are a wide variety of methods for solving Markov Decision Problems, as they are very common in the real world. For example chess and checkers are MDPs and so are most other board games, particularly those where the player can see everything about the current state of the game by looking at the board. Chess and checkers are very familiar games to us, and most people have a great deal of contextual knowledge ranging from psychology to logic to military strategy that they can draw on when playing these games. Machines are not normally so lucky, and usually have to work with little or no prior knowledge. As Russell & Norvig (2003) explains, a machine's view of the problem is more like: Imagine playing a game whose rules you don't know; after a hundred or so moves, your opponent announces "You lose". This is reinforcement learning in a nutshell. Surprisingly, even with such difficult problems posed to them, RL algorithms have had considerable success.

3.3 Value functions

As Russell & Norvig (2003; §21.1) explains, the best way to solve an MDP is to learn an optimal policy for behaving in the environment, and then follow it. A policy is simply a mapping from every state in the environment to the optimal action to take in that state. The logical way to learn a policy is to learn the value of a utility function $U(s)$ for each state s, or an action-value function $Q(s,a)$, giving the expected utility of taking a given action in a given state. An optimal utility agent can use its utility function to select the optimal action at each step, and therefore follow an optimal policy. This is done by using a model of the environment to find those states adjacent to the agent's current position and choosing the action leading to the state with the highest value. The utility values the agent uses are typically the discounted future reward (DFR) that an agent following an optimal policy would expect to receive from that point onward, although other models of optimality are possible (Kaebling, Littman, & Moore, 1996). Formally, the utility function that the agent aims to learn is:

$$U^*(s_t) \leftarrow E\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k}\right\} \qquad \text{Eqn. (4)}$$

The function $U^*(s_t)$ is the expected value of the sum of $\gamma^k r_{t+k}$ for all k from zero to infinity. The variable γ is the discount factor and r is the scalar reward value. In other words, the utility agent attempts to learn the DFR it expects to receive by behaving optimally, starting from a state s at a time t. In contrast, an action-value agent, learns a function $Q(s_t, a_t)$ that describes the DFR the agent will eventually receive (by continuing to follow the optimal policy) if it takes action a in state s. The optimal policy in this case is just to choose the action with the maximum q-value in the current state. The q-function is therefore:

$$Q^*(s_t, a_t) \leftarrow E\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k}\right\} \qquad \text{Eqn. (5)}$$

When the values are correct, $\max_a Q(s_t, a_t) = U(s_t)$ so utility based and action-value learning are able to converge on the same behaviours. However, there are specific benefits and drawbacks to each approach, and there is no absolute rule for deciding which to use (see

Russell & Norvig, 2003). In general, the main drawback of learning a utility function is that the agent also needs a transition function to know what the next state is, while the action-value agent does not. It must either learn the model, which takes longer and is difficult, or be given one, which puts a greater burden on the programmer. On the other hand, the action-value agent will end up having to learn redundant utility values (if there is more than one way to get to a state, it will have to learn a value for each), which is slower and takes more memory. Additionally it cannot perform look-ahead search because it has no way of knowing what the next input/state will be (though the utility agent might not either, if the values are not deterministic). On the other hand, it is the simplest, and sufficient for our purposes.

3.4 Learning algorithms

The fundamental methods used to learn these value functions are dynamic programming, Monte Carlo methods, and the method of temporal difference. All of the methods can solve reinforcement learning problems, but they have their individual strengths and weaknesses.

Dynamic Programming (DP) is a well established collection of algorithms that can be used to compute optimal policies for a Markov Decision Problem. Because they need a perfect model of the environment to function they are called ***model based algorithms***. The classical DP algorithms, including policy evaluation and policy iteration, are broadly considered the foundation of reinforcement learning (Sutton & Barto, 2005), and the later methods can be viewed as attempts to achieve much the same effect, only with less computation and without assuming a perfect model of the environment.

More sophisticated model based algorithms do not start with the assumption of a perfect model of the environment, but learn the model through experience of the environment, and then use that to compute the optimal policy. These include certainty equivalent methods and adaptive dynamic programming (particularly with the prioritised sweeping heuristic), which are explained fully in Russell & Norvig (2003). In general, model based algorithms make more efficient use of the data they collect at each step, and so they are able to learn good policies in fewer steps than model based environments. This makes them suitable for environments where it is relatively expensive, difficult or slow to collect new data, such as in embodied agents.

Model-free algorithms also start with no model of the environment, but instead of learning one they aim to determine the value function directly through experience (Schmidt, 2005). They are generally simpler to implement and work at least as well as alternative methods in simulated environments, so they are used for the continual learning experiments in this research.

Monte Carlo methods are an established group of model free algorithms, but they can only be used for episodic tasks, and are therefore undesirable for experimenting with continual learning. The other basic model free algorithm, temporal difference (TD) learning, is effectively an approximation of dynamic programming that avoids the need to learn the environment’s transition model. It therefore requires far less computation per step, which makes it more tractable in large spaces.

The essential idea of TD learning methods is that an agent’s predictions of how to achieve the best outcome will become more accurate over time (as the outcome grows nearer). This means that the agent can use its newest predictions to update older predictions in a process called bootstrapping, instead of waiting until the end of an episode to integrate its experiences into a predictive function. Bootstrapping is particularly important for lifelong learning agents, as real life cannot easily be divided into episodes, and without bootstrapping these agents would be unable to learn anything at all. This is an important reason for using TD methods in continual or lifelong learning research.

3.4.1 Temporal difference learning

To build a policy using TD methods, the agent only needs to explore the environment and use TD-updates to incorporate the feedback it receives at each step into its state-value function. The basic TD update has the form:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \qquad \text{Eqn. (6)}$$

This just means that the last prediction $V(s_t)$ is modified by the reward r plus the difference between the current prediction $V(s_{t+1})$ (representing the future reward, discounted by γ) and the last prediction. The magnitude of the update is moderated by a constant α. The current prediction should be more accurate than the last one, and so the value function should eventually converge to $U^*(s_t)$ and the agent can behave optimally.

All of the fundamental dynamic programming, Monte Carlo and temporal difference methods were developed from well-justified mathematical theory. However, TD learning is unique in that it appears to have biological analogues in human and animal brains. One example is from Schmidt (2005), who found that the activity of midbrain dopamine neurons in reward-related learning has properties very similar to temporal difference reinforcement learning. It is hard to say exactly how relevant or important this is in the context of building artificially intelligent learning agents, but it is encouraging that this type of algorithm can scale up to the enormously complex learning tasks faced by real biological brains.

3.5 Control

The basic algorithms that use TD methods to control an agent are Adaptive Heuristic Critic (AHC) and Q-learning. In effect, they ensure that the agent continues to explore its environment and improve its policy whilst simultaneously exploiting its existing knowledge of the environment value function to maximise its return while converging. Both algorithms can be used to learn good policies but Q-learning is easier to implement and has the theoretical advantage that it is proven to always converge on an optimal policy (though this does not apply in the general case where function approximation is used). This property of convergence is documented in Baird (1995).

Q-learning agents learn a value function $Q(a, s)$, which approximates the discounted future reward (DFR) that the agent predicts it will receive by performing action a in state s. So in the Wumpus world, moves leading to the goal should have a high value, and moves leading to the Wumpus should be low value. The update rule in Q learning is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \qquad \text{Eqn. (7)}$$

This update is calculated whenever an action a_t is executed in state s_t leading to s_{t+1} and resulting in a reward r_{t+1}. The values of future rewards should also be discounted by a factor γ, which describes the agent's preference for current over future rewards. Because our problem formulation assumes an infinite horizon for decision making, meaning that we care about significant events arbitrarily far into the past, future rewards would tend towards infinity if we did not discount them. If that happened we would be unable to compare states (since every state would be either plus or minus infinity) and further learning would be impossible.

Once the agent has learned the function (or optimal policy) correctly it can simply use a one-step look ahead to choose the best action to take. However, while the agent is still learning, it has a problem. The agent's learned model of the environment is different to the environment itself, so actions it predicts will lead to the highest reward may not do so in reality. If the agent always chooses the best action (called greedy behaviour) then it may ultimately receive suboptimal rewards. More importantly, the agent may fail to find new states in the environment that lead to an even higher reward. To avoid this problem, the agent needs to actively explore the environment. It is often sufficient to simply have the agent behave randomly a small proportion of the time during learning, although more sophisticated schemes are possible.

3.6 Learning with Approximation and Generalisation

The problem described in section 3.2 is called a discrete MDP because there are a finite number of states in the environment and the agent's transitions between them are instantaneous (in that the agent's position can never be described as partway between two states). As there are only a small number of states in that problem it is sufficient to use Q-learning with a simple lookup table to store the value of each state, and visit each state in the environment until we are convinced we have found a sufficiently good policy.

However in many problems (including the real world), environment states are not discrete, and it is possible to be in any of an infinite number of states. One early and widely studied RL problem often implemented in real world agents is called the cart pole problem, pictured in Figure 9.

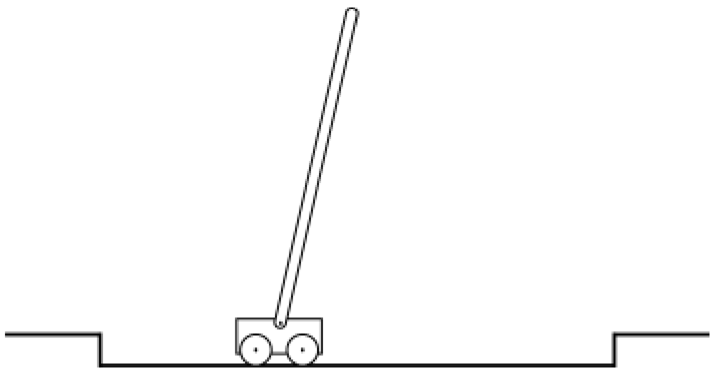


Figure 9: The cart pole problem, as described in Sutton & Barto (2005; §3.3)

The goal in cart pole is simply to keep the pole balanced upright for as long as possible by driving the cart back and forth on a track. In this case the actions available to the agent are discrete (pulse the motors left, right, or do nothing) but the environment has infinitely many states (described by the angle between the pole and the horizontal, and the position of the agent on the track).

In this case it is clearly impossible to represent the Q-function over all possible states and actions in the environment using a lookup table because that would require an infinite amount of storage. Instead we are forced to make do by visiting some finite subset of the infinite variety of states, and using that data to generalise to new states as we encounter them. This type of generalisation is an example of function approximation and it is a type of supervised learning. Generalisation is a complex topic that is studied in several fields, including machine learning, artificial neural networks, pattern recognition, and statistical curve fitting. For the most part, however, we only need to work out how to modify RL techniques to draw upon existing research, such as the neural network function approximation introduced in Chapter 2. There are some additional difficulties as Boyan & Moore (1995) noted, since the value functions of RL agents using generalisation are no longer guaranteed to converge to the optimal policy in some cases. However, they also noted that those cases can be avoided by problem-specific hand tuning. Additionally, Baird (1995) found that it is possible to use generalisation and still guarantee convergence in general, by slightly modifying the learning algorithm with his residual gradient technique.

While generalisation is essential in continuously valued environments, it can also be very useful in complex or large discretely valued environments. Possibly the most famous example of this is Garry Tesauro's TD-Gammon system (Tesauro, 1994). TD-Gammon used TD learning with nonlinear function approximation using a multilayer neural network in a manner similar to the system described in §3.3. There are approximately 10^{20} discrete states in backgammon (Kaebling, Littman, & Moore, 1996, p. 270) and the game tree has a branching factor of around 400, making it too large for the conventional heuristic search methods used successfully in games like chess and checkers at the time. However, using TD learning and generalisation, TD-Gammon learned to generalise by playing 300,000 games of backgammon against itself, while visiting only a small fraction of the total state space. Impressively, Tesauro's system learned to play backgammon competitively against world-class human players.

3.6.1 Tile coding

The simplest way to add generalisation to a temporal difference agent is to employ a linear method such as tile coding to learn the value function. The non-generalising agent uses a table with one entry for every possible state in the environment, which as noted above is impractical for large and infinite spaces. However, by using tile coding we can still use the same linear structure. Essentially all we have to do is divide the state space up into a finite number of non-overlapping regions or "tiles" that exhaustively cover the space. Then we simply associate every possible state in the tile with a single value (expected DFR, as above). For example, in the cart pole problem above, we could use the tiling shown in Figure 10. In this scheme, every possible configuration of the cart and the pole can be described by a pair (corresponding to a region of the state space), for example $A1, A2, \dots, B1, B2, \dots$ and so on, with the current state being E2. Now we can easily store a value for each region using a small one or two dimensional array. Then we can carry out TD learning as usual, just by performing updates on the entire current region instead of just the current state.

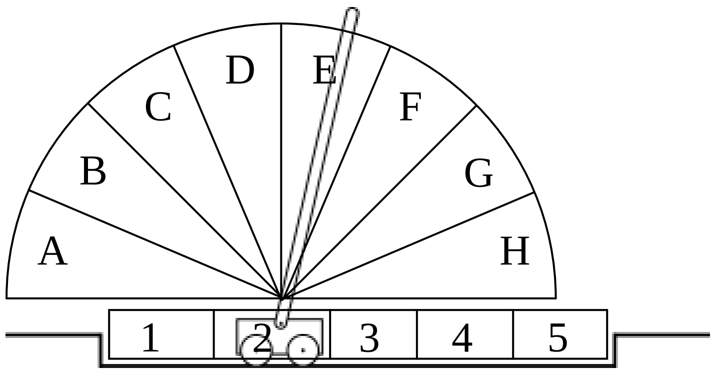


Figure 10: A sketch of a tile-coding for cart pole

Figure 10 shows the agent in state (E2). In this state the agent would likely learn to pulse the motors right, but in state A5 it would learn to pulse left. After learning, we could expect the agent to have learnt to move left in states including A-C pole positions, and right in F-H positions.

3.6.1.1Limitations of tile-coding

The simple tile coding scheme above is more than sufficient to solve the cart pole problem. However, single cart pole is a simple problem and is considered a standard test that any generally useful dynamic control system should be able to pass. The current state of the art in embodied agents is a so-called variable universe adaptive fuzzy control algorithm (Hongxing, Jiayin, Yundong, & Yanbin, 2004) that is able to successfully balance a robotic cart pole system with a quadruple articulated arm, physically constructed and tested in the researchers' laboratory. This feat is almost certainly far beyond the reach of tile coding Q-learning. It may be possible to perform the task if a more sophisticated function approximation system were used, but it would still be very difficult, as Lee & Perkins (2008) found when trying to solve the triple jointed cart pole problem.

The main problem with tile coding is that it only really works well if you can ensure that for each tile, every state within that tile has roughly the same value, or at least the same optimal action. This means that the value function either needs to be very smooth or we need a good idea of the shape of the value function before we solve the problem. It is also possible to randomly try a variety of hashing schemes until we find one that works, and we can divide the space into a larger number of tiles if we use the trick of hashing tiles to reduce memory requirements, as Sutton & Barto (2005; §8.3.2) suggests. However this can only take us so far, after which we must turn to a more intelligent generalisation system to handle more complex tasks.

3.6.2 Other methods of generalisation

One way of providing more powerful generalisation capabilities to a TD agent is to use a neural network to approximate the value function instead of a table or array. The standard way of achieving this is described in Lin (1991). Since this is the method used in later sections of this work, it is described in detail in §3.8.

Another type of generalising RL algorithm is called G-learning, described fully in Chapman & Kaelbling (1991). G-learning is a standard Q-learning system that uses a decision tree for generalisation. Effectively, this allows the agent to calculate the best way to tile the state space and how large to make the tiles, an approach that is far more efficient than choosing a tiling randomly or by hand. It also has an advantage over neural network generalisation because we do not need to choose the network architecture before learning. Chapman and Kaelbling found G-learning worked very well on a range of problems, even on problems where an agent with a neural network for generalisation failed to find the optimal policy. However, they also found that there are some environments to which generalisation using backpropagation is especially well suited, such as the one used in Lin (1991). They also found that G-learning is unsuited to the same problem, due to the nature of the decision tree learning algorithm.

It may be possible to obtain some of the best of both worlds by using a system that combines the advantages of both decision trees and neural networks, such as the one presented in Rountree (2007). In his thesis, Rountree developed a method for initialising a multi-layer perceptron (feed-forward neural networks are MLPs) by first training a decision tree on the input examples. Then he applied another algorithm to use the knowledge contained in the decision tree to set the initial architecture and connection weights of an MLP. The resultant network trained significantly faster than a standard MLP. His methods were only tested on supervised function approximation problems, but they should also be applicable to reinforcement learning by autonomous agents. However, at the present time there does not appear to be any published research investigating this possibility.

3.7 Non-Markov environments

The learning methods described above are widely used to solve Markov Decision Problems, and some of them are even mathematically guaranteed to solve MDPs optimally under reasonable conditions.

However, not all real-world problems are Markovian. In many real-world problems it is not possible for the agent to know exactly what state of the environment it is in. There might be some underlying variable invisible to the agent that nevertheless affects the parts of the environment that the agent can directly perceive. Other times, the agent's current sensory input from the environment (or perception[6]) may be limited, distorted, or ambiguous, so that it is only possible to know exactly where the agent is by examining the steps leading up to the current perception, as shown below.

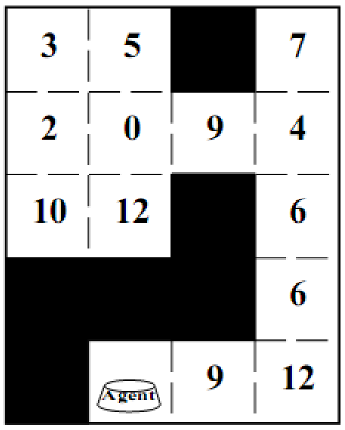


Figure 11: A grid environment with ambiguous sensory input, adapted from Ring (1994)

In this example, the only input the agent receives is the number on each square, representing the configuration of the walls around that square. For example, a 6 means there are walls to the east and west, but not north and south. If the agent receives a 9 as input, it cannot be sure which state it is in. However, if it remembers seeing a 0 or 4 just previously, then it can be sure it is in the northernmost 9 state. For reference, the basis for the numbering scheme used here is described in full in Figure 17.

[6] The agent's current (possibly limited or distorted) sensory inputs are called a 'perception' of the environment. In a Partially Observable Markov Decision Problem, the agent's goal is to disambiguate the current perception, allowing it to see the exact state of the environment.

In either case the agent needs to solve a Partially Observable Markov Decision Problem, or POMDP (Kaebling, Littman, & Moore, 1996, pp. 267-269). These problems are an active area of research because standard learning methods, including the Q-learning method described in Section 3.5, require complete observability to learn or they may oscillate around inadequate solutions. Another reason is that it is generally much harder to work with POMDPs than MDPs. In fact, Russell & Norvig (2003, p. 627) states that even finding approximately optimal policies to complex POMDPs is PSPACE-hard[7] (that is, so hard that finding the optimal solution for problems even with only a few dozen states may be infeasible).

CHILD (Ring, 1994) is one system that attempts to solve temporal POMDPs by determining the context for the current perception. It does so by building 'context units' that work as a short term memory, enabling it to draw on perceptions from the arbitrarily distant past to help disambiguate the present input, and therefore behave correctly.

3.7.1 Delay lines

The simplest general method for solving POMDPs is to use a technique called delay lines (Ring, 1994, p. 26) that allow the agent to disambiguate its perceptions. The technique is widely used in temporal problems, but due to its simplicity it is not always referred to by that name, as in Schmidt (2005, p. 26). Delay lines achieve the same goal as the Temporal Transition Hierarchies (TTH) method that CHILD uses, but in a simpler fashion. With the delay lines technique, the agent simply keeps a window of the past N perceptions in a buffer and presents them to its learning system simultaneously, so that all of the necessary contextual information is directly available.

If the agent is using a neural network as a function approximation with an input representation of M units then we only need to modify it to have N*M input units, and then encode the contents of the perception buffer into the input layer of the network in order. The key advantage of this method is that it is really only a clever way of encoding a state vector, so it works with existing non-temporal algorithms without modification. This is much simpler than a whole new architecture like TTH and is still sufficient for many purposes. However, it has some disadvantages (not present in TTH) that prevent delay lines from being an ideal general solution. Firstly, we need to correctly choose how many previous inputs the agent will remember to disambiguate its state. Secondly, we need to remember the entire representation of the entire set of perceptions, even if it turned out that only one or two features of the input

[7] PSPACE is a computational complexity class greater (that is, more complex) than the well-known NP complexity class, of which the Travelling Salesman and 3-SAT problems are members.

were necessary to provide the context. Third, the agent has no way of changing or controlling which perceptions to remember. If the correct behaviour for the current state depended on a single perception 100 steps ago then the agent would need to store all of the last 100 steps to learn it, whereas a TTH agent would (in theory) need to keep far less information. However, none of the environments Ring (1994) used to test CHILD require a memory of more than the last one or two states to learn a successful policy.

3.8 Q-Learning with neural network-based function approximation

The backpropagation algorithm (described in section 2.2) works well for supervised learning problems, but we may also wish to apply neural networks to the task of solving MDPs in an RL context. Just as we can improve on the basic TD or Q-learning algorithms by using tile coding or a decision tree to represent the value function in those algorithms, we can use a neural network to achieve generalisation in the value function of an RL agent. The agent architecture necessary for this is shown in Figure 12.

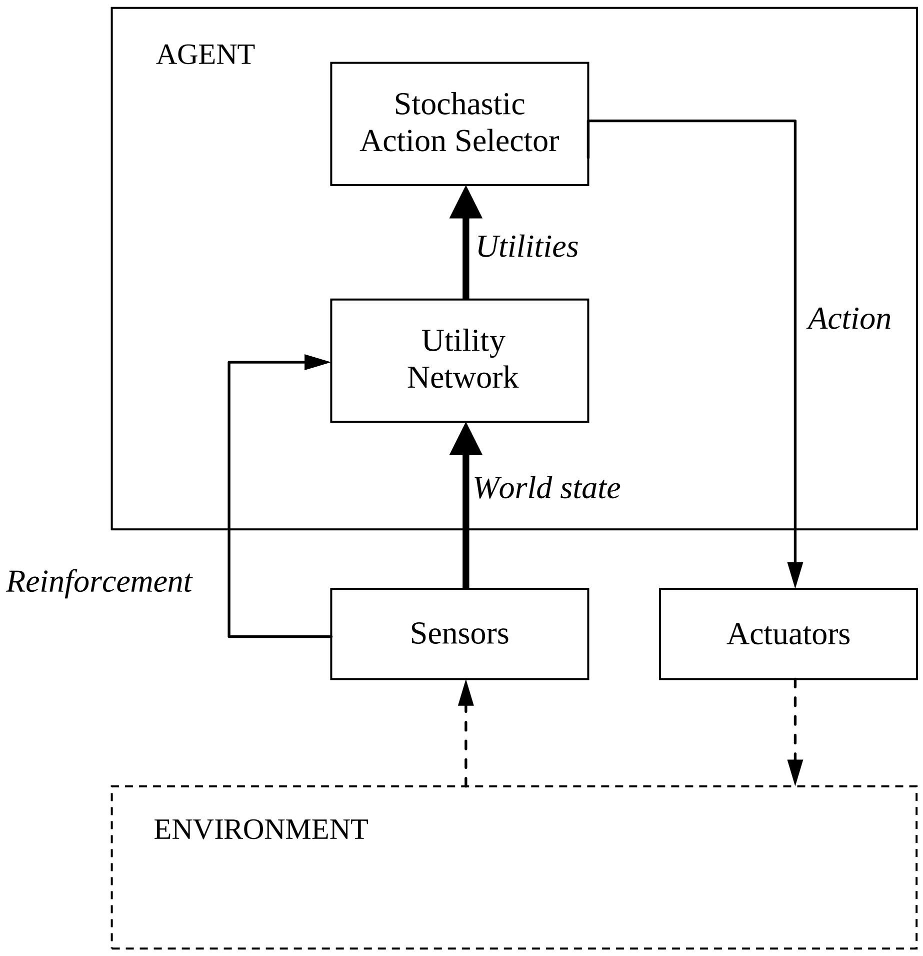


Figure 12: A model of the interactions in a Q-learning agent system, adapted from Lin (1992)

The stochastic action selector is only necessary during learning, to allow the agent to explore the environment and discover a more accurate value function. Instead of just choosing the action with the highest utility ("greedy" action selection, which can be used after learning), actions are chosen randomly according to a probability distribution determined by the utilities of the actions, using the formula in Equation 8.

$$Prob(a_i) = e^{m_i/T} \Big/ \textstyle\sum_{k=0}^{i} e^{m_k/T} \qquad \text{Eqn. (8)}$$

Where the variable m_i is the utility of action a_i and the temperature T adjusts the randomness of action selection. This optimisation (as in a simulated annealing search) ensures that the network will eventually converge on the optimal solution (although this may require infinite training) where a greedy search may get stuck in a local minimum of the state space (Shang & Wah, 1996). Lin suggested using an individual network for each possible discrete action available to the agent. His idea is that this should reduce representational overlap and therefore prevent the values of different actions in the same state from interfering with each other (Lin, 1992). However, using separate networks also isolates the knowledge they each contain, and that may hurt the overall efficiency and generalisation capability of the agent. An alternative solution would be to use an algorithm such as Context Biasing to reduce representational overlap, or pseudorehearsal to reduce forgetting. In any case, the algorithm for Q-learning with a neural network approximating the action-value function works as outlined in Figure 13.

```
1. $x \leftarrow current\ state;$
   $U_i \leftarrow util(x, i), \qquad for\ each\ action\ i;$
2. $a \leftarrow select(U, T);$
3. $Perform\ action\ a;$
   $(y, r) \leftarrow new\ state\ and\ reinforcement\ signal;$
4. $Adjust\ utility\ network\ by\ backpropagating\ error\ \Delta U$
   $through\ it\ with\ input\ x, where\ \Delta U_i = \begin{cases} U_i' - U_i & if\ i = a \\ 0 & otherwise \end{cases}$
5. $go\ to$ 1.
```

Figure 13: A generalising Q-learning algorithm

In short, the steps are: find the utilities of each action, select one according to equation 8, perform the Q-learning update, backpropagate the difference between the old and new predictions, and repeat. The $util(x, i)$ step is calculated by executing the current state/observation vector on the network (the forward pass of the standard feed-forward backpropagation network training regimen). It also sets the network up for the backpropagation step which occurs after an action is chosen by $select(U, T)$, where U is the set of predicted DFRs and T is the temperature variable. A noteworthy point about this method is that it only works for problems where there are a finite number of discrete actions available to the agent, for example to move North, South, East or West. This model can also be adapted to continuous actions by pairing the state and an action in that state as the input to the network, and then having the output (a single continuous node) represent the value of taking that action in that state.

Learning with a neural network in this way has some subtle but important differences to supervised training. The update is only a single backpropagation step, so the environment/DFR pair is not trained until the error is reduced to near-zero, as it would be in batch learning (normally used in supervised MLPs). In other words, this scheme must use an online learning neural network. It is possible to use a batch-update neural network in a Q-learning system, but it cannot be used to learn the Q-values directly from the environment. Instead, a lookup table must be used as a short term cache to collect Q-value pairs from the agent’s explorations until there are enough to teach them to the network. An example of this system is Rivest & Precup (2003). One possible reason why this alternative is not widely used is that catastrophic forgetting problems seem to be far more severe than with the former system. In fact, the cited authors found their agents often had terrible performance unless the cache was made so large that the agent was effectively just a tabular learner; they speculated that the problem could be catastrophic forgetting.

3.8.1 Catastrophic forgetting is still an issue

It happens that the situation described in Section 2.3 (also called *conceptual drift*) is exactly the problem inevitably faced by a neural network when it is used as a value function approximator in the system described above. The input-output pairs taught to the network are generated by the Q-algorithm from the agent's experiences of the environment and the network's own value predictions, which are initially just random nonsense. Logically it follows that the initial mappings presented to the network will be wildly different to the mappings generated by the Q-algorithm when the agent has had a chance to thoroughly explore the environment. The reason for this is that initially the network has received no feedback for taking any action in its current state, so it will predict a small, random positive or negative DFR. When the agent has reached the goal state a few times and has had a chance to incorporate that prediction into the current state and action, the DFR will be very different, and the network has to be taught the new value. Recently, Baddeley (2008) theorised that a standard MLP's inability to handle conceptual drift may be a significant problem for Q-learning MLP-type systems that could slow learning and even prevent it entirely.

The already serious problem of handling conceptual drift in this type of system is naturally compounded when we try to perform serial learning with an RL agent. One reason why we might try to do this is if we want to teach an agent to behave in several similar but distinct environments, with different optimal policies. Another reason is to try to emulate the technique of *shaping* (described at the beginning of Chapter 3). An example of useful serial learning would be if we had an autonomous robot vehicle that we wanted to teach to drive on windy mountain roads (slowly and carefully) but also on high-speed motorways, where the same basic driving skills are needed but a different speed, level of caution and so on, are called for. Ring's (1994) main result is another example of a sequential MDP-learning task.

3.9 Summary

Chapter 3 introduced the second class of machine learning problem: reinforcement learning (RL), and the subclass of Markov Decision Problems (MDP). The framework and theoretical model by which RL problems are solved was briefly introduced. Several different reinforcement learning algorithms were explained; including Q-learning (which is used in the next section). Variations on the basic MDP were discussed, such as problems with continuous inputs and outputs, and temporal problems, or Partially Observable MDPs. Finally, the details of how to combine Q-learning with the neural network algorithms of Chapter 2 were explained, and the practical consequences of learning by this method.

4. Investigation

Chapter 2 examined how neural networks can be used to solve supervised machine learning problems. It also explored how neural networks can perform poorly when used for online or continual learning, and introduced the main types of algorithms used to improve performance in those scenarios. Chapter 3 introduced the class of reinforcement learning problems, and also the possibility of employing neural networks to solve them. Reinforcement learning problems are the type of problems that this research is primarily dedicated to solving, since they are the type most often encountered by intelligent learning agents in the real world. Similarly, we are interested in solving those problems using neural networks for memory and generalisation. This is partly because less powerful algorithms may not scale to RL problems large enough to be useful, and partly because we know that memory systems resembling neural networks are successfully employed by intelligent agents in the real world. Finally, we are interested in solving sequential RL problems, because intelligent agents clearly need to be able to solve many sequential, related, and unrelated problems over the course of their existence.

Surprisingly, given this context, the problem of catastrophic forgetting in neural networks has almost exclusively been studied in relation to supervised learning. However, there has been some related research in the RL community. Rivest & Precup (2003) noted that catastrophic forgetting could be damaging the performance of their RL approximation algorithm, but did not perform a systematic investigation. Baddeley (2008) found that preventing catastrophic forgetting improved the performance of a neural network Q-learning algorithm on a single reinforcement learning task, but did not investigate the class of lifelong learning and long term recall problems that this research is interested in exploring. Ring (1994) did investigate continual and lifelong RL problems, but used a novel and unconventional[8] type of linear constructive network to solve them, instead of the common and well understood Feed-Forward Back Propagation network (FFBN) we intend to use. With all of that in mind, this chapter is an attempt to investigate the problem of lifelong learning in relation to reinforcement learning, using the type of Q-learning agents with neural network function approximation introduced in section 3.8.

[8]Ring also noted that the network is not biologically plausible, despite its effectiveness.

4.1 RL-Glue

The experiments carried out in this investigation were for the most part written in Java, using the RL-Glue framework. RL-Glue is a highly modular, multiple language capable distributed framework for carrying out research on the subject of single agent reinforcement learning and in programming contests (Tanner, 2008). The basic framework is organised around sets of interchangeable agents, environments, and experiments, each of which have no direct contact with the others. Instead, the components pass messages to and from the RL-Glue framework, which coordinates their interactions, as shown in Figure 14.

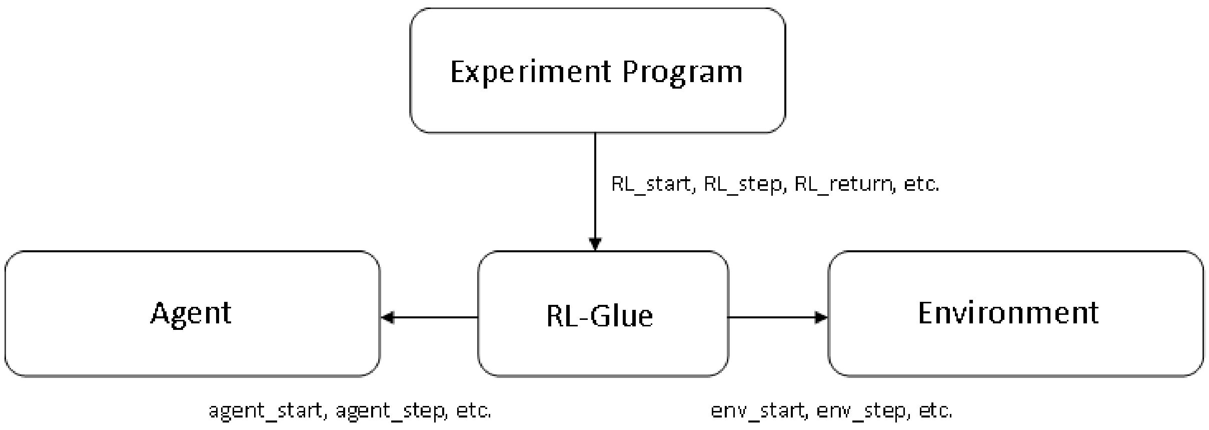

Experiment Program
RL_start, RL_step, RL_return, etc.
Agent
RL-Glue
Environment
agent_start, agent_step, etc.
env_start, env_step, etc.

Figure 14: RL-Glue high-level architecture, adapted from Tanner (2008)

In this scheme the experiment component acts as a script for a run, the environment as a stage, and the agent the performer. Time passes as a series of discrete steps taken in response to decisions made by the agent, and the agent's input and feedback is received from the environment as an observation of the current state. If the environment is fully observable then the observation received by the agent will simply be the current state of the environment. However, this is not the case in a POMDP, so the framework passes around observations (or perceptions) instead of states.

4.2 Environments

Given the framework in Section 4.1, the major components of this investigation are the environments, the agents that operate within them, and the experiments that make it possible to measure their effectiveness. Two different environments were used, in order to separately study the project goals of Section 1.2.

4.2.1 Sequential maze environment

The environment used to investigate this question was broadly analogous to the dataset Robins (1995) used in his experiments with pseudorehearsal in supervised agents. Those experiments started by teaching a neural network a sizeable base population of random input-output mappings, and then trained the network completely on one new random mapping at a time. As each pattern was learnt, the error of the network on the base population was measured to determine the extent of the forgetting.

Similarly, this RL environment consists of a base population (a large maze) and incremental units of new knowledge (small mazes). Of course this is not an exact translation of Robins' supervised data to an RL context, but it is close enough to allow us to explore the same concepts. The key differences are that the Robins' mappings were entirely independent of each other and presented in random order with uniform frequency. The states in this experiment are naturally grouped by their adjacency in the maze, and states 1, 8, 56, and 63 will be visited much less often than states 28, 35, 36, and 43.

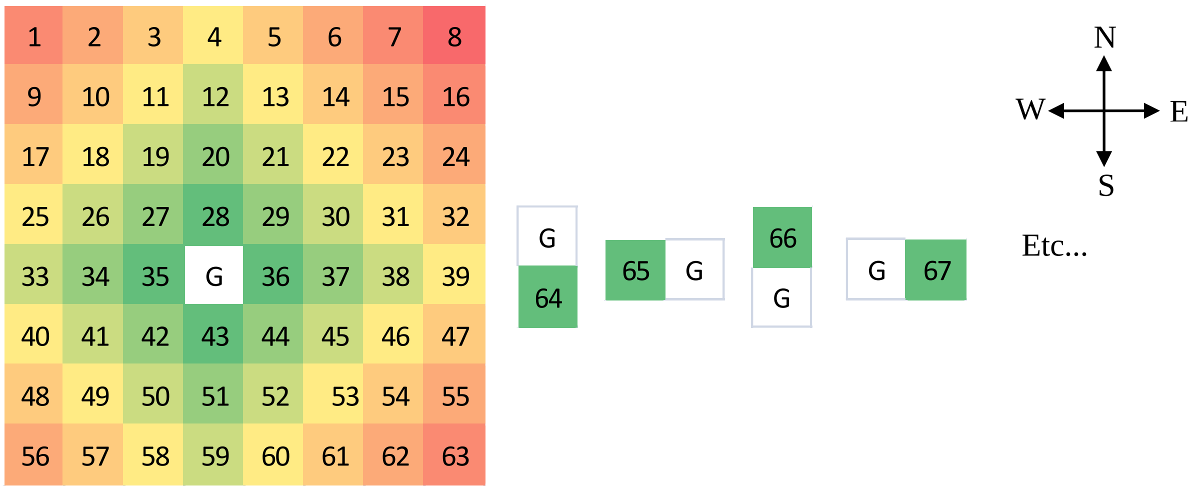


Figure 15: A problem consisting of a set of simple mazes to be solved sequentially

The sense perception that the agent receives at each step in this environment is a random unique fixed-width binary integer mapped to the unique identifier for each state. There are four possible output actions, representing a decision to move north, south, east, or west (no diagonal moves are allowed). Note that even if there are only two or three sensible moves to make (as in a corner or against a wall of the base maze), or even only one move (as in a mini-maze), all four actions are available to the agent, and it is not obvious to the agent (without look-ahead or a transition model, which the Q-learning agent does not use) which moves will fail. If the agent attempts to move somewhere inaccessible, i.e., off the map ("bumping into a wall") it remains in the same state but does not receive any indication that the movement failed. When the terminal goal state is reached a reward of +1.0 is given, or +0.0 otherwise, with a limit of 50 steps allowed before terminating the episode with no reward.

These mazes are deliberately trivial, and represent roughly the smallest decision an agent can learn to make, i.e., "in state X take action Y". Because we are using Q-learning, we would expect a successful agent's solution to the first miniature maze to look like $[64] \rightarrow [1.0, 0.9, 0.9, 0.9]$ (if 0.9 was the discount factor for future rewards). Since the maximum number of steps is large relative to the maps' size, a random agent or an agent with a stochastic action selection policy should be able to consistently reach the goal in the environments within the maximum number of steps without having to learn anything. Testing is therefore performed with deterministic (greedy) action selection enabled so that agents will have to learn some sort of useful mapping in order to reliably find the goal. Note that the colours are an indication of the distance from each cell to the goal, and therefore show the relative value that the agent will assign to each cell (or rather, moves that will take it to that cell).

An important point about this environment is that it effectively makes a standard rehearsal scheme impossible. If rehearsal was done with actual input and output pairs, the output portion would always be zero, except for the actions that lead directly to the goal. If the network value function learned environment feedback directly it could not behave optimally, because it would view all actions that do not directly and immediately lead to a positive reward state as equally useless. In order to capture the DFR values that the agent needs to remember to perform optimally, we have to ask the agent for its DFR predictions for each actual state of the environment. The resulting input-output pairs are pseudoitems, because the outputs do not correspond to actual feedback from the environment. Additionally, the neural networks in the agents used in the later sections of this chapter all have 4 output units, one for each possible action it could take. To generate complete input-output items from the environment, we would have to collate examples of the feedback received by taking each action in each state to be rehearsed, which would be unwieldy (and still non-optimal, because the network's goal is to learn DFRs, not to learn feedback directly).

4.2.2 Continual learning environment

The second RL environment used in this research is a replica of the continual mazes environment from Ring (1994), instead of the sequential maze problem used in the previous sections of this chapter. Ring's CHILD agent performs well on this test, so it makes for a fairer comparison against the other agents we developed. It is also a good example of a continual reinforcement learning problem, as opposed to the more common episodic RL problems (such as cart pole, mentioned in Section 3.6). Figure 16 shows a diagram of the continual mazes that were adapted to the RL-Glue interface.

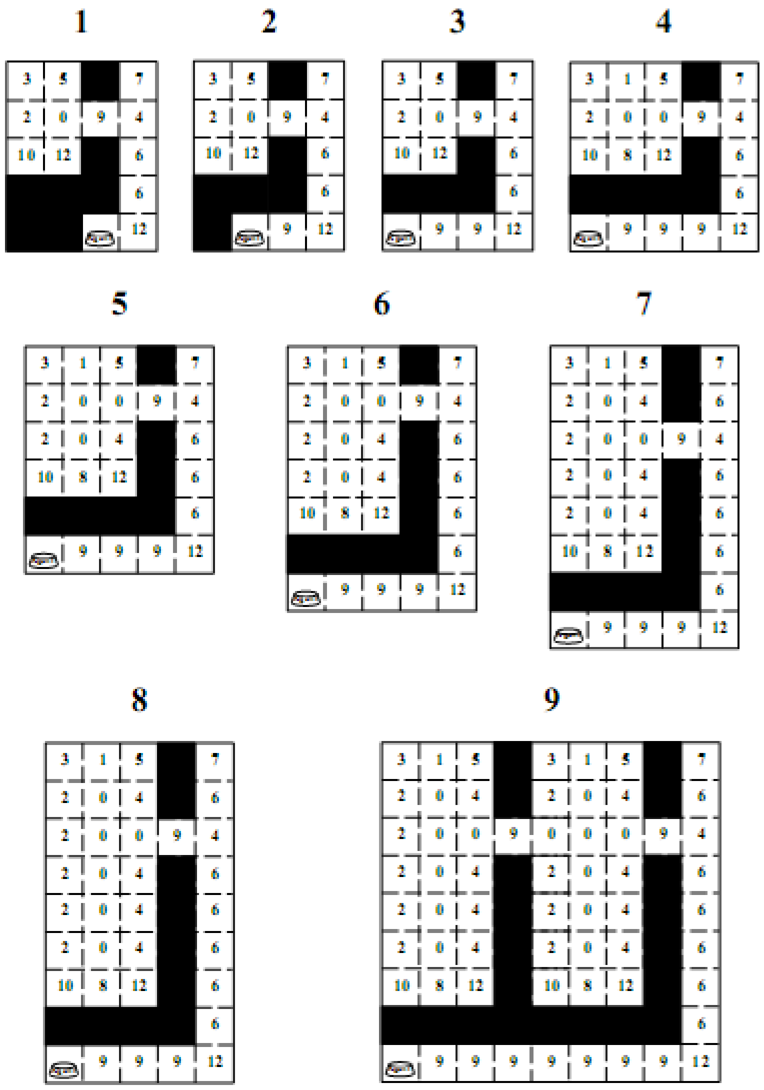


Figure 16: A continual learning environment series, adapted from Ring (1994)

The input the agent receives in each cell is simply the number shown on that cell, and moving and bumping into walls and obstacles is handled exactly as it was in the first environment (Section 4.2.1). These mazes are plainly more complicated than the deliberately simple mazes in Section 4.2.1, have many obstacles and more problematically for the agent, ambiguous sensory input. The numbering of the cells follows a rule described by the cells or obstacles in the neighbouring cells, described below:

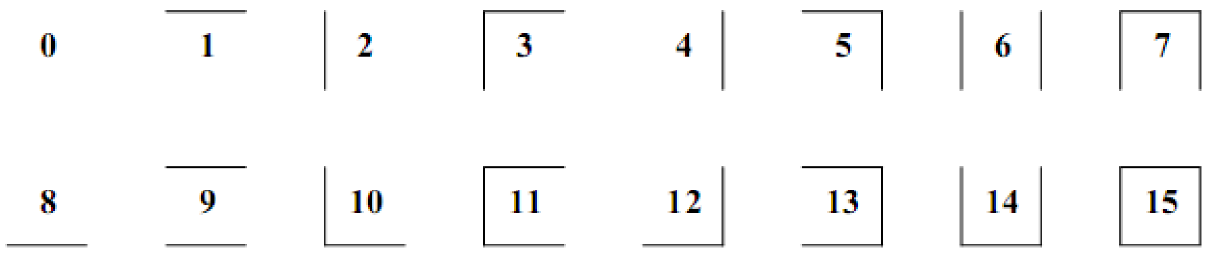


Figure 17: Description of agent senses

Because this is the only information provided to the agent it must learn to distinguish the identical sensory perceptions to determine the best action to take in all positions of each maze. The logical way to do this is to take into account a history of where the agent has recently been to determine the probable current state. Although any two identical sense states must also have identical neighbouring states (given the definition in Figure 17), the states adjacent to the neighbouring states are not usually identical in both cases, and this allows the agent to disambiguate the current state.

The mazes are deliberately similar, and each one is only gradually more complex than the last. The intention of this design is that the optimal value function for each maze should have considerable overlap with that of the previous maze. A successful continual learning agent should therefore be able to transfer its existing knowledge to the new task, and in doing so speed up learning.

4.3 Agent design

Because these experiments were designed to compare a number of different schemes for continual learning in scalar reinforcement problems the continual learning systems were all based on a single modular Q-learning reinforcement agent. This agent uses the scheme described in Figure 13, but instead of performing the execution and backpropagation steps on the neural network directly it communicates through an interface to a supervised learning algorithm, which is chosen dynamically and initialised at runtime.

The generalised algorithm for this agent is the same as the neural network specific one, except that the update and execute functions are generic, and implemented by each supervised learning algorithm class according to their type. Any type of algorithm can be used as long as it can support basic operations such as execute, update, clone, and reset. The temperature, discount factor, and learning rate parameters are therefore part of the Q-learning agent, while algorithm-specific configuration such as the number of hidden units to use are part of the dynamic algorithm object it contains.

A feed-forward neural network, Ring's Temporal Transition Hierarchies, a simple tabular algorithm, and a random number 'algorithm' (for benchmarking and testing) were implemented according to their descriptions in Chapters 2 and 3 and tested in the course of this research. In this section, special consideration is given to the pseudorehearsal and context biasing agents, since they are relatively novel compared to the other agents implemented in the course of this research.

4.3.1 Pseudorehearsal

The pseudorehearsal learning agent is a reasonably straightforward addition to the basic neural network agent, which was implemented more or less exactly as described in Section 3.8, by implementing a neural network version of the generic value function component for the modular Q-learning agent described above. Once the neural network agent is in place, adding pseudorehearsal to enable continual learning is just a matter of generating pseudoitem pairs and learning them alongside the normal environment/reward pairs. Despite this simplicity, there are a number of possibilities, optimisations, and variables to account for, and some important differences to the pseudorehearsal algorithm developed for supervised learning problems.

The best all-around performer of the pseudorehearsal schemes[9] Robins (1995) tested was sweep pseudorehearsal, followed closely by random pseudorehearsal, and this research uses schemes that are similar to both. In both of his systems and our own, a collection (buffer) of pseudoitems is selected or generated every time a backpropagation is performed on an actual item. However, in Robins' system the actual item remains the same until it is fully trained, whereas in our system the actual item must change every step, as the agent navigates around the environment.

Another matter requiring consideration is the method by which new pseudoitems are generated, and how often. The possibilities are:

1. A large batch of pseudoitems, generated only once, after the initial base population (i.e., the first maze of either environment above) is learned
2. Generate a batch at some other logical interval, such as when a new environment is entered or a new task is started (sweep pseudorehearsal)
3. Generate just enough pseudoitems to fill the buffer every time the agent takes a step (random pseudorehearsal)

All three possibilities have different benefits and drawbacks.

The first method has the least computational overhead (from generating new pseudoitems) relative to ordinary backpropagation, but it has the conspicuous flaw that anything learnt after the initial population is learnt will be subject to catastrophic forgetting. Also, many problems will not have a large initial population that absolutely must be remembered anyway. However, it makes some sense to try this in problems such as the one described in Section 4.2.1, which was deliberately designed to be similar to the experiments of Robins (1995).

The second method should be slower by some degree, but it makes a great deal more sense as nothing important should be forgotten. However, it works best when there is a sensible interval to use to decide when to generate new pseudoitems, and there needs to be some sort of signal from the environment to tell the agent to refresh the buffer. An alternative algorithm could generate new items every time the agent has taken some arbitrarily or empirically chosen number of steps, but there is not usually any guarantee that the agent will learn things at a strictly linear pace (since it moves in a non-deterministic way), so it may be hard to find a reliably good interval. It can be made to work in problems like those described in Section 4.2.1 at least, and it might also be good enough in other situations.

[9] The rehearsal schemes (which had similar performance) were ruled out due to the problems with selecting and caching mappings for rehearsal in an environment with significant conceptual drift, described in Section 2.4.1.

Finally, the third solution is the most generally applicable, and it guarantees that everything the agent learns will be used at some point by the rehearsal mechanism. However, it is also the slowest, as so many more pseudoitems will need to be generated. Performance measured in steps taken may also be slightly worse, because the pseudoitem pairs may reinforce partially learned or temporarily incorrect information captured by the network. Robins (1995) found random pseudorehearsal to be generally less effective.

There are also the additional questions of how large the batches and the active buffer should be, and whether it is better to always backpropagate the actual item before the pseudoitems, or if randomly interleaving actual items and pseudoitems has some benefit.

4.3.2 Context biasing

Context biasing fits readily into a Q-learning neural network agent, because all we have to do is add in the weight adjustment step before the normal backpropagation for each step. However, the same issue occurs as with pseudorehearsal, where the order in which training pairs are received from the environment in reinforcement learning is a lot less predictable than in supervised learning. As previously mentioned in Section 2.4.4.1, context biasing does not do well on standard tests of recall, so in the studies reported in Section 4.7 an additional test of French's savings measure was introduced to measure its performance.

4.4 Experimental design

The experiments measure the performance of a continual learning agent along three different metrics:

1. Ability to retain knowledge while acquiring new information
2. Ability to relearn information that is naturally lost over time in any finite system
3. Ability to generalise, by exploiting knowledge transfer to perform better in new situations

Baddeley (2008) investigates another metric; namely, the speed of convergence of a MLP agent on learned values. He concludes that catastrophic forgetting is a significant handicap in the learning process of a MLP agent, and that a pseudopattern rehearsal strategy may be quite beneficial.

The experiments described in the following sections test the essential memory and learning characteristics that an intelligent agent needs to possess. Additionally, comparing several continual learning systems allows us to determine the best way to solve sequential reinforcement learning tasks.

4.5 Investigation of pseudorehearsal

An important question that must be addressed is whether a neural network solving an RL problem will have a noticeably different learning profile to a network solving supervised problems. In particular we are concerned with whether it will be subject to catastrophic forgetting to a greater or lesser degree, and whether the pattern of sudden degradation is similar despite the data being structured in a significantly different way and presented in a different manner.

4.5.1 Experimental procedure

The goal of this experiment was to determine if catastrophic forgetting is a problem for a basic Q-learning neural network agent, whether pseudorehearsal could alleviate the problem, and what variations of the rehearsal algorithm work best. The procedure for measuring the impact of catastrophic forgetting on learning agents was therefore as follows:

1. Learn the base population maze
2. Learn a new mini-maze
3. Test and record the performance of the learning agent on the original maze
4. Repeat steps 2 and 3 for 32 more mini-mazes in random, non-repeating order (without resetting the agent)

The experiment was repeated 50 times with a new agent each time (i.e., a new instance of the agent class with a newly initialised state but with all of the same parameters). The process for learning a maze is to start the agent from a random position in the maze, allow it to explore the maze until it reaches the goal or until 50 steps have passed (the mazes are small so this limit is quite reasonable), and repeat, periodically testing its deterministic performance on the maze, until it passes perfectly or it eventually fails to converge (after failing the test 50 times). For efficiency, the initial maze was tested at a long interval (100 trials), while the smaller mazes were tested after every learning trial.

An agent's performance on a maze is tested by first disabling its stochastic action selection[10], then starting it from every valid position in the maze in turn and recording whether or not it reaches the goal from each position within 50 steps. Its performance is the fraction *successes / (successes + failures)*. This simple test setup was performed with a variety of agent types and configurations and produced a range of results.

[10] The effect of this is to make the agent always choose the action with the highest predicted DFR, i.e., it will behave as a greedy agent and make no attempt to explore the environment

4.5.2 Results

Figure 18 shows the performance of a Q-learning agent using a standard three-layer perceptron with 32 hidden units while executing the learning task in section 4.5.1 in the environment depicted in Figure 15 (except for Figure 19, all figures in this section refer to that problem exclusively). The agent used a learning rate of 0.1, a discount factor of 0.9, and momentum of 0.5, and the trial was repeated 50 times to ensure a representative result. For this test the network has 7 binary input units (for example, the network input 0000101 corresponds to state 5), which are needed to represent the 96 state inputs in binary, and 4 output units. Each of the four outputs corresponds to a prediction of the DFR the agent expects from taking that action in the current state (importantly: not the actual reward values, which are usually just zero), as explained in Section 3.8. The vertical bars attached to each data point show a single standard deviation above and below the mean to indicate the variation in the different instances of the trial.

For the RL experiment, average chance performance is approximately 0.3 when the tests are carried out with a neural network agent with an entirely random set of fixed mappings (i.e., the performance of a completely untrained agent, without stochastic action selection). This line of effectively zero performance is marked in bold in the figure.

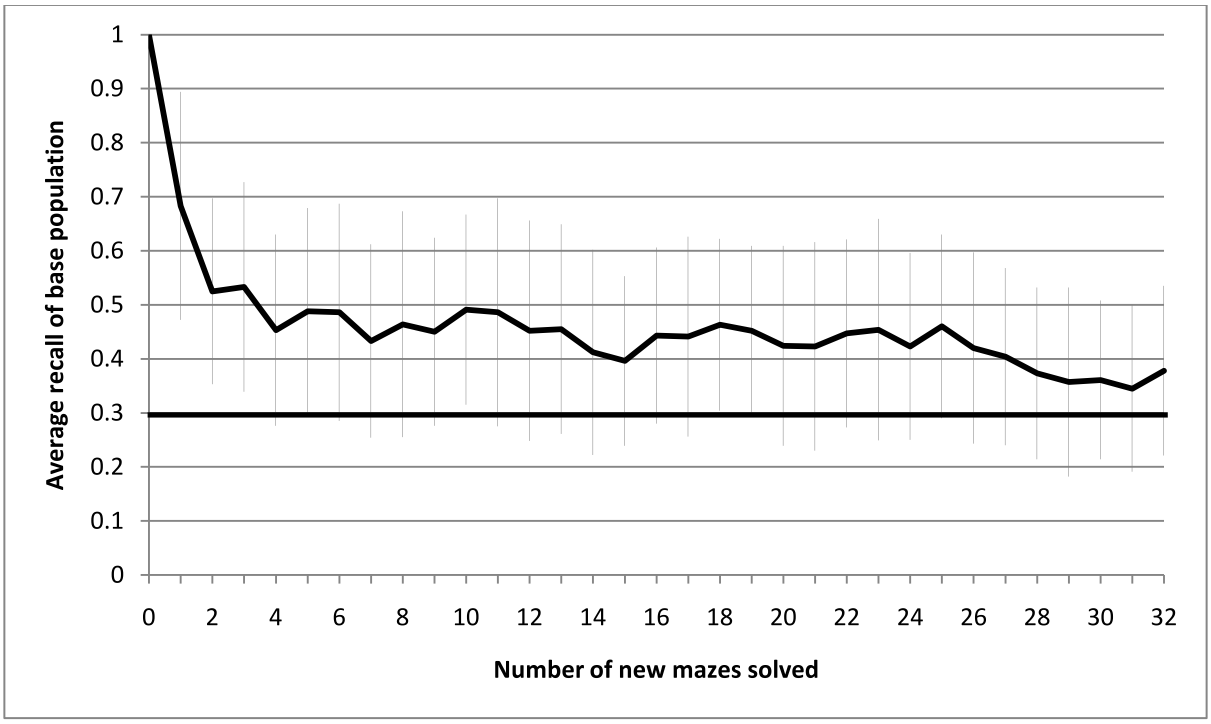


Figure 18: Measurement of recall for a basic neural network agent learning new items

This result is comparable to Robins (1995) results with an unmodified neural network attempting to recall an original dataset while learning new information. Those results are reproduced below, for comparison.

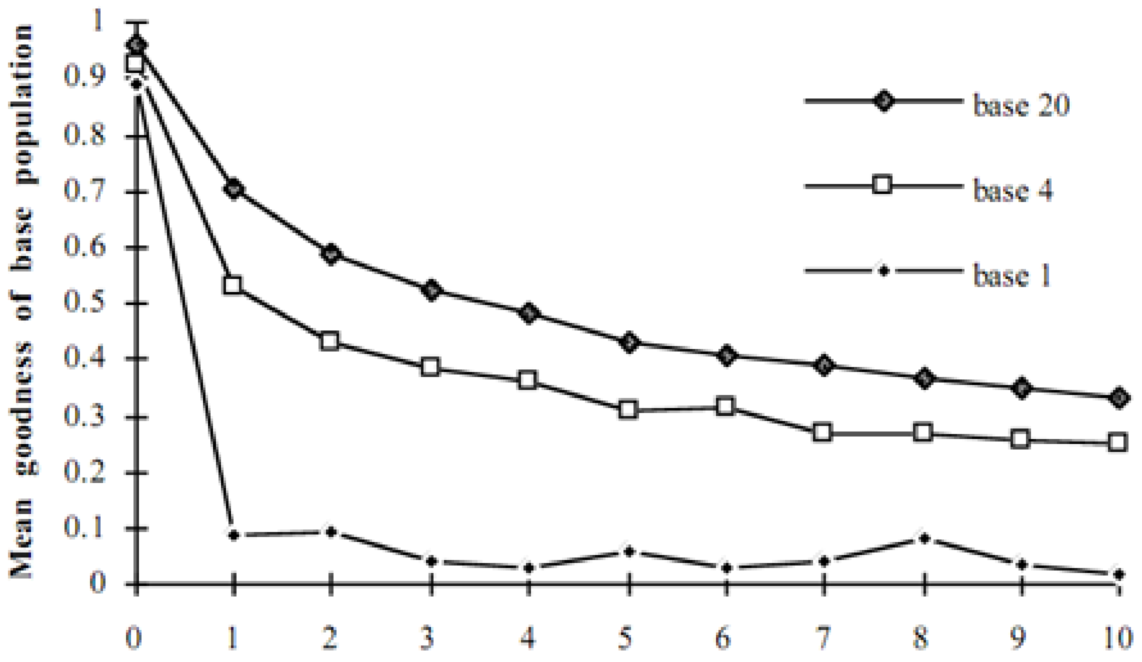


Figure 19: Fall in base population goodness as new items are introduced (x-axis), adapted from Ring (1995). Figure shows results for base populations sizes (number of patterns) of 1, 4, and 20.

The results from the Q-learning trial appear closest to the "base 4" series in Robins' supervised trial, although exact numerical comparison is impossible because the experiments used different measures of performance, different problems, and Robins' results are normalised between chance and perfect performance (as opposed to ranging from zero to perfect performance, since in practice performance is always above zero in his test). However, the trend is very similar. The Q-learning agent's performance drops to around 0.5 very quickly and then only gradually declines as new items are introduced. This result shows MLP Q-learning agents are at least as susceptible to catastrophic forgetting as supervised MLP learners, if not more so. However, it is still possible to avoid the problem, as the results in Figure 20 show. Note that the purpose of this test was to establish the limit of best possible performance that could be achieved. As such, the parameters used are not necessarily practical or realistic for larger problems.

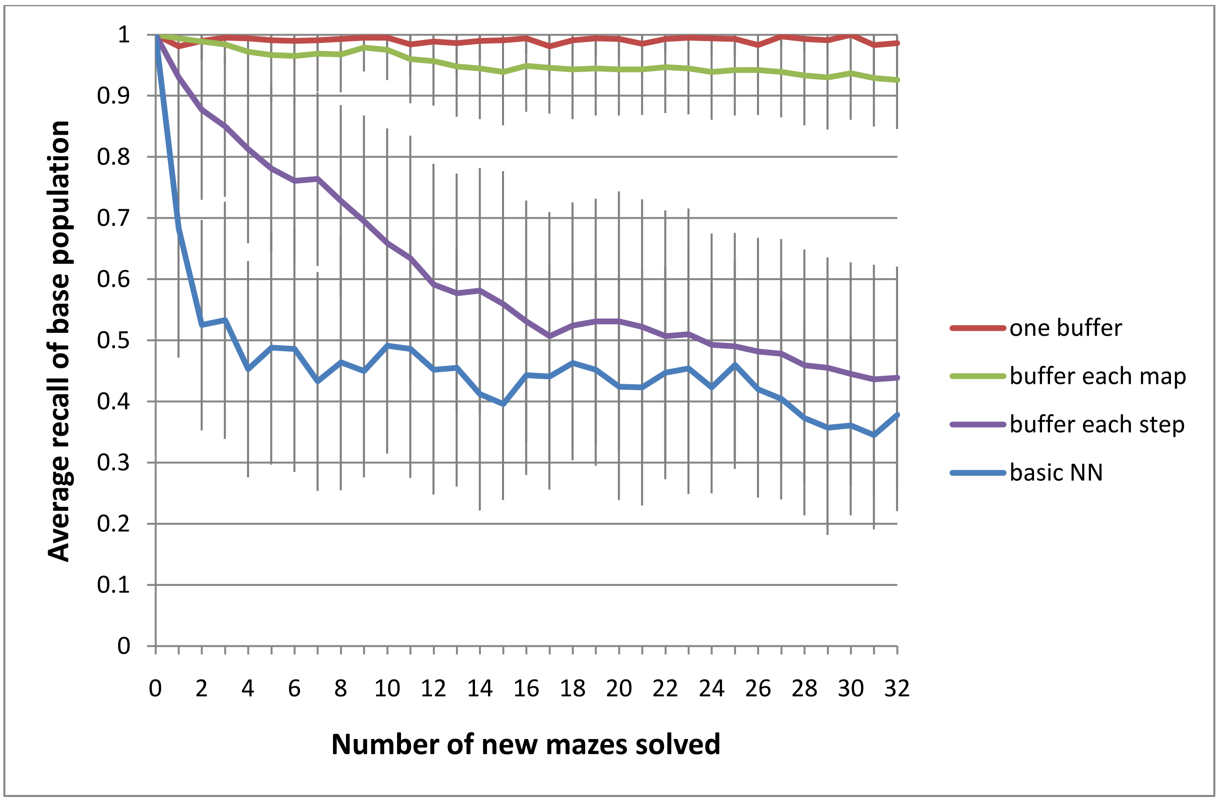


Figure 20: Performance of agents using pseudorehearsal to avoid catastrophic forgetting

Figure 20 shows the performance of a Q-learning agent with a neural network modified to use pseudorehearsal. Three different configurations are shown (alongside the non-pseudorehearsal agent, for reference), the difference being how often the pseudorehearsal buffer is regenerated from the network's stored knowledge. The other parameters were the same as for the unmodified network, and the vertical bars are also one standard deviation, although there is considerable overlap. In the once only (the "one buffer") configuration, the pseudo-population was generated only after the base maze had been solved, since generating it immediately would only force the agent to rehearse the random function represented by its initial weight configuration.

There are 63 states in the base population, so one pseudoitem was generated for each sense input the agent encountered in the maze and all 63 were rehearsed along with each new sense input the agent received thereafter. In this section we refer to this as "*full pseudorehearsal*", even though it would be possible to generate additional pseudoitems for inputs the agent did not encounter, and for states that the network input layer can represent, but which the environment cannot generate (i.e., anything between 96 and 127). The performance is already perfect for this agent, and the other agents showed no improvement with larger numbers of pseudoitems[11].

These results show that catastrophic forgetting of a base population[12] can be completely avoided by using pseudorehearsal, since the base recall performance of the "one buffer" agent still averages nearly 1.0 even after all of the new mazes have been solved (learned). However, it appears that for the algorithm to work perfectly you must be able to rehearse with as many pseudoitems as there were discrete states in the base population, be able to store all of those pseudoitems indefinitely, and you must not care about catastrophic forgetting in everything learned after the base population (because the other series in the figure do not have perfect performance). None of these provisions are entirely reasonable for large problems, so it is worthwhile to investigate compromises that may still allow acceptable reinforcement learning performance.

The other two series in Figure 20, "buffer each step" and "buffer each map", show an agent using a completely new pseudo-population after each new item is learned, and an agent using completely new pseudoitems every time they are needed (at every step/transition where the agent receives feedback from the environment) respectively. The former has the advantage that it captures all of the new knowledge acquired by the network with minimal effort spent generating pseudoitems, while the latter has the advantage of never needing a long term memory store outside of the network. Unfortunately, the continual learning performance of the "buffer each step" agent is nearly as bad as the unmodified agent, so some external memory is clearly required. The performance of the "buffer each map" agent (which effectively has a separate short term memory) is reasonably good, remaining over 90%. However, for many continual RL problems it is not obvious exactly when a new 'item' has been learned, so some other way of periodically refreshing the pseudo-population store is required. The other logical option is to refresh the memory after some fixed arbitrary number of steps (call it N). Figure 21 shows the performance of identical agents using different values of N.

[11] Note: "full pseudorehearsal" is still pseudorehearsal, because the output portions of the input-output rehearsal items comes from the network, instead of being cached from actual feedbacks received from the environment.
[12] In this case, the base population is the *state* → *DFR* utility value function the agent learned while exploring the large beginning maze.

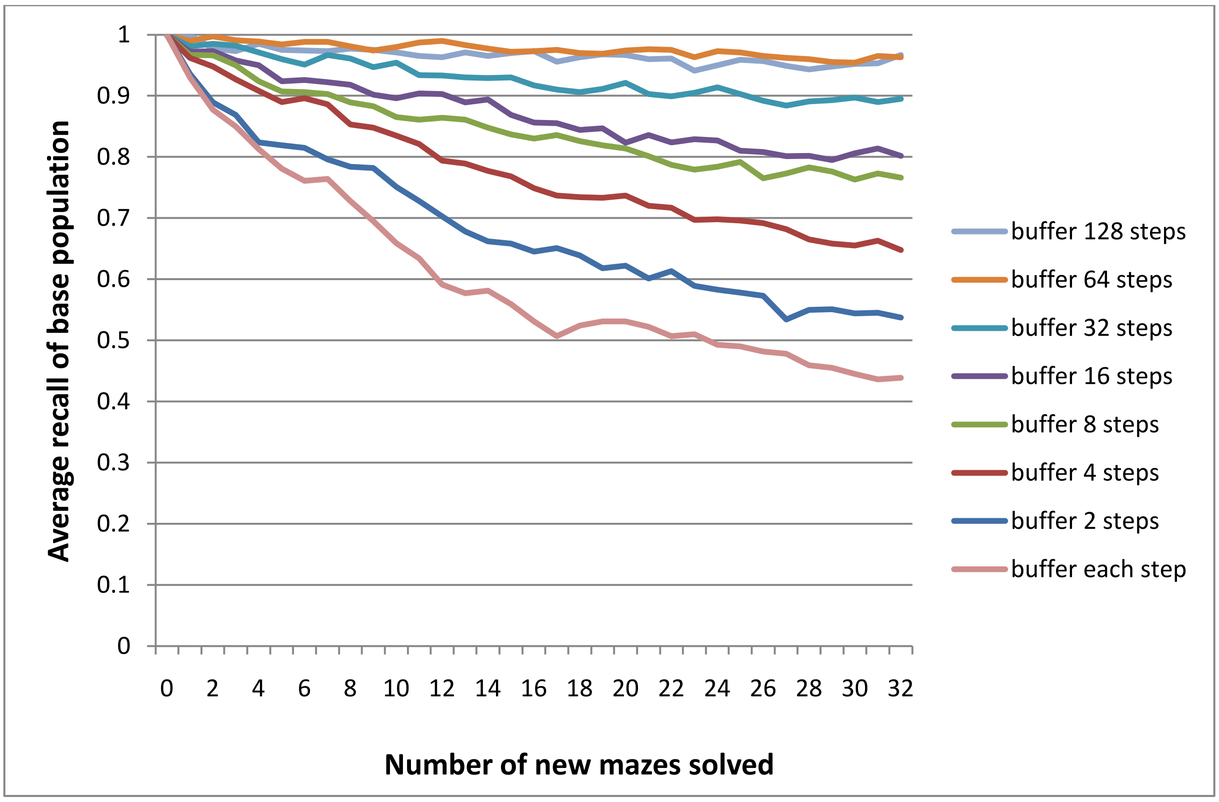


Figure 21: Performance using pseudorehearsal with a new pseudo-population every N steps

Figure 21 shows the results of testing with full pseudorehearsal and a range of values for N. To help summarise these results, Figure 22 shows the final recall performance of each agent plotted as a single line. As Figure 21 shows, this metric roughly orders agents by their overall performance. Note that each episode or trial run of the agent exploring the maze has a maximum number of steps, after which the trial is terminated, and N can be greater than that number. When to regenerate the maze is based on the number of steps the agent has taken during the current "learning phase" (intervening these phases are the "test phases" where the performance of the agent set to use a deterministic action selection function is used to decide whether it has learned the maze).

There is a clear relationship between the refresh period and continual learning ability. The performance of N=32 is comparable to the 'each new map' method, while anything more converges on the performance of the single pseudo-population agent[13]. However, the each map method is still theoretically preferable, because then it is certain that nothing important escaped the pseudorehearsal process, and any pseudoitems corresponding to new items will

[13] This is because the delay until the second pseudo population is needed (N steps) will eventually be longer the total length of the test, for increasing values of N.

represent ‘completely learned’ knowledge, rather than half formed or temporarily incorrect predictions.

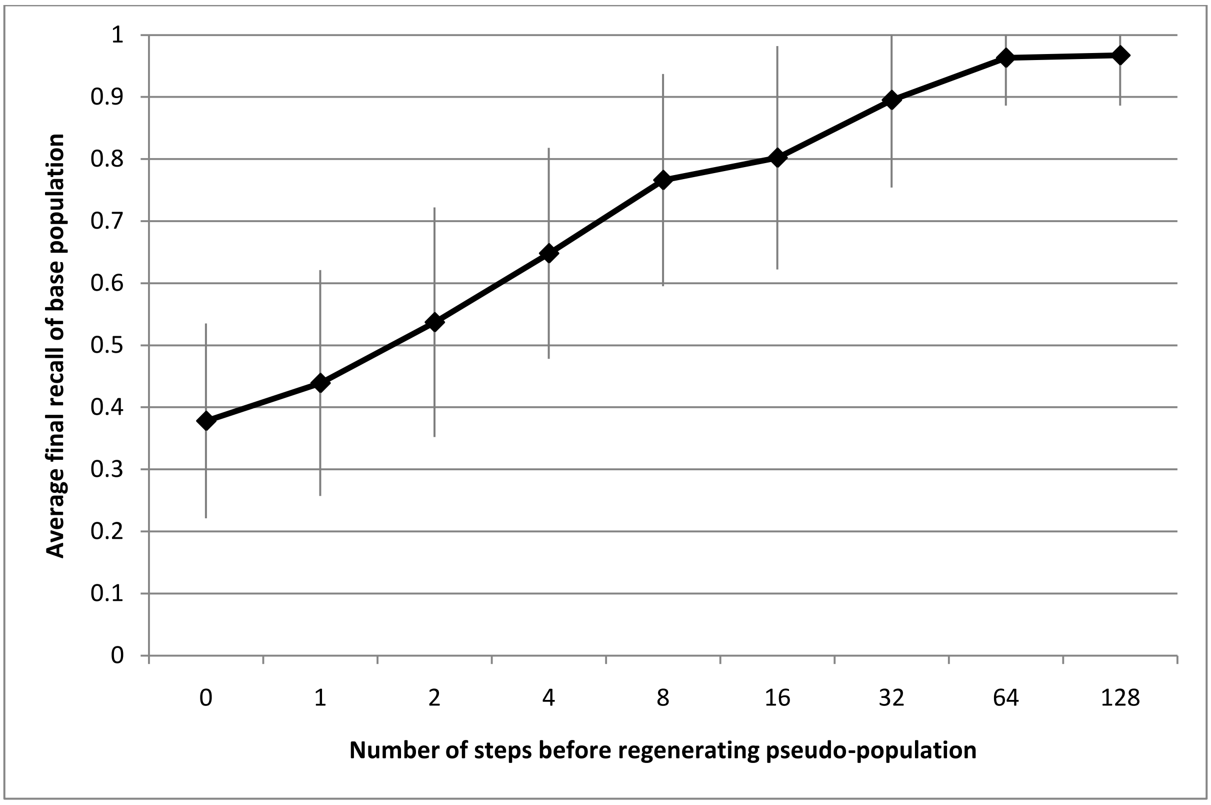


Figure 22: Final recall performance using pseudorehearsal with a new pseudo-population every N steps

4.5.3 Full-pseudorehearsal performance

Generating a pseudoitem for every state the network encounters (and actively rehearsing all of them every time the agent incorporates feedback about a new prediction) is effective, as the previous section shows. However, it is also inordinately time-consuming, and impractical for any problem with a very large or infinite (continuous) number of states.

Figure 23 shows the average number of steps taken by the agents during learning, while Figure 24 shows the average time taken for the different configurations to finish a trial, including time taken to create and rehearse pseudoitems. The time measurements were made using total CPU thread time used by the agent process in order to avoid random interference by other applications, and were repeated (50 times, as before) to ensure consistency. The exact times taken should only be considered instructive only relative to each other, since their parameters and programming are not perfectly optimised and the machine running them is not overly powerful.

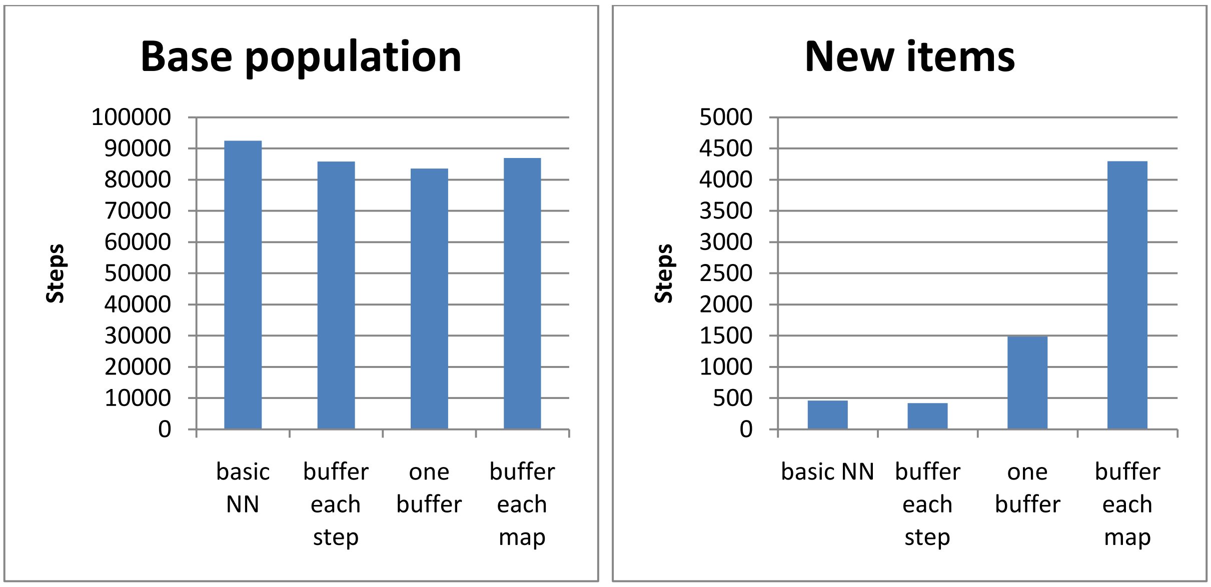


Figure 23: The average number of movement steps taken by different agent configurations in each trial

As Figure 23 shows, the agents using pseudorehearsal need between two and 8 times as many steps as the basic neural network agent to complete a trial, when the base population steps are excluded (because that number will always be approximately the same for all of the agents).

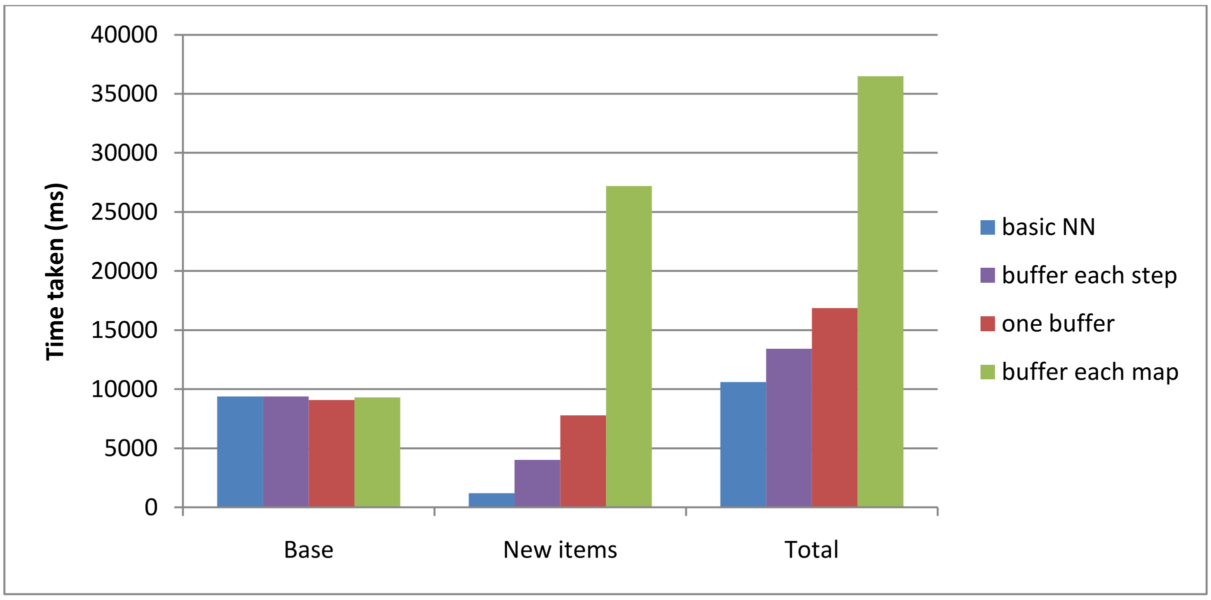


Figure 24: The average CPU time taken for different neural network agent configurations to complete a trial

The agents using pseudorehearsal need even longer to complete the new mazes than the unmodified agent, when the difference is measured in milliseconds. This is because each step takes longer due to the rehearsal operations. In fact, when using full pseudorehearsal it takes longer to learn a few independent items (separate mini-mazes) than it does to learn a large set of interdependent ones (i.e., the large base maze).

The number of steps is so much greater because the sheer amount of rehearsal being performed at each step interferes with the actual data that needs to be learned. Additionally, the ‘buffer each map’ agent takes so many more steps than the single population (‘one buffer’) agent because that agent needs to rehearse a larger set of items (the once only agent has a set of 64 pseudoitems instead of up to 96 when the 32 mini-mazes are included), and the extra items interfere with learning even more. However at the same time, the agent using all-new pseudoitems each step does not take that much longer, despite using the maximum possible number of pseudoitems, because those pseudoitems interfere with actual learning less. This is because the more recently the items were generated, the smaller the difference between them and the current shape of the network value function (this is also the reason why that agent is so poor at preventing catastrophic forgetting).

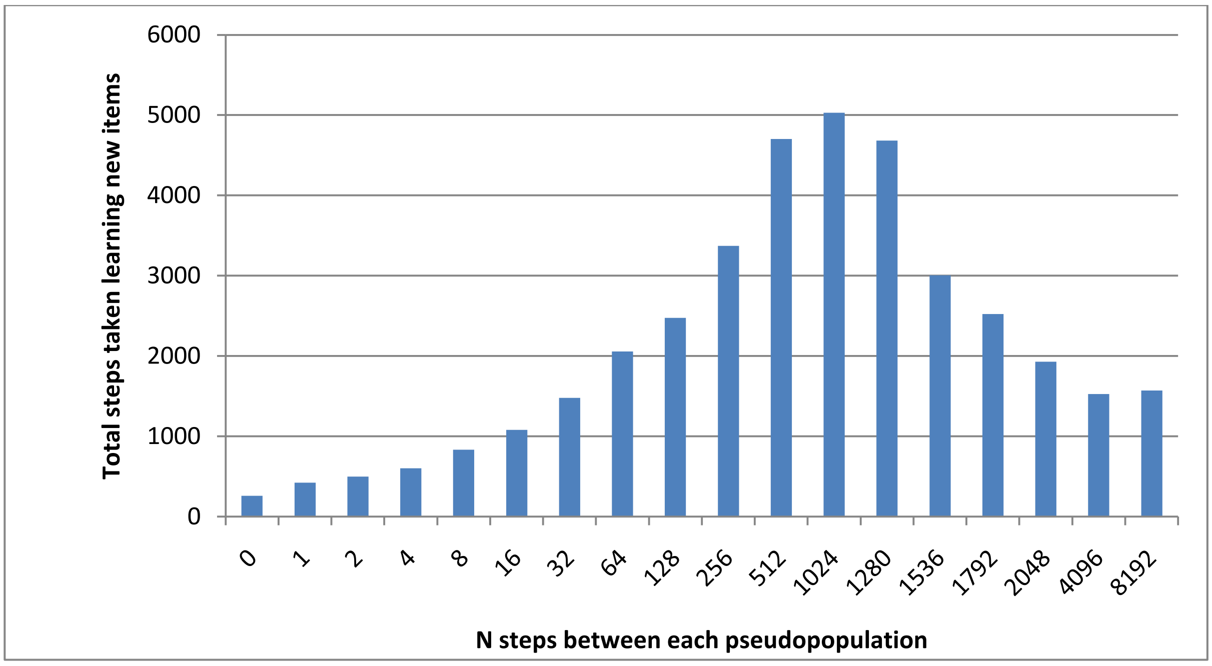


Figure 25: The number of steps taken while learning the new items, with an agent using a new pseudo population every N steps

Figure 25 demonstrates the continuum generated by the tension explained above, by plotting the average number of steps taken by an agent using a new buffer every N steps, for a range of values of N. Performance steadily worsens as the pseudoitem data becomes further removed from the data coming in from the environment (on average) as new mazes are encountered, until the balance tips in favour of a smaller, less changeable dataset. Figure 26 (included for reference) is an expanded version of Figure 22, showing a continuation of the same trend as before.

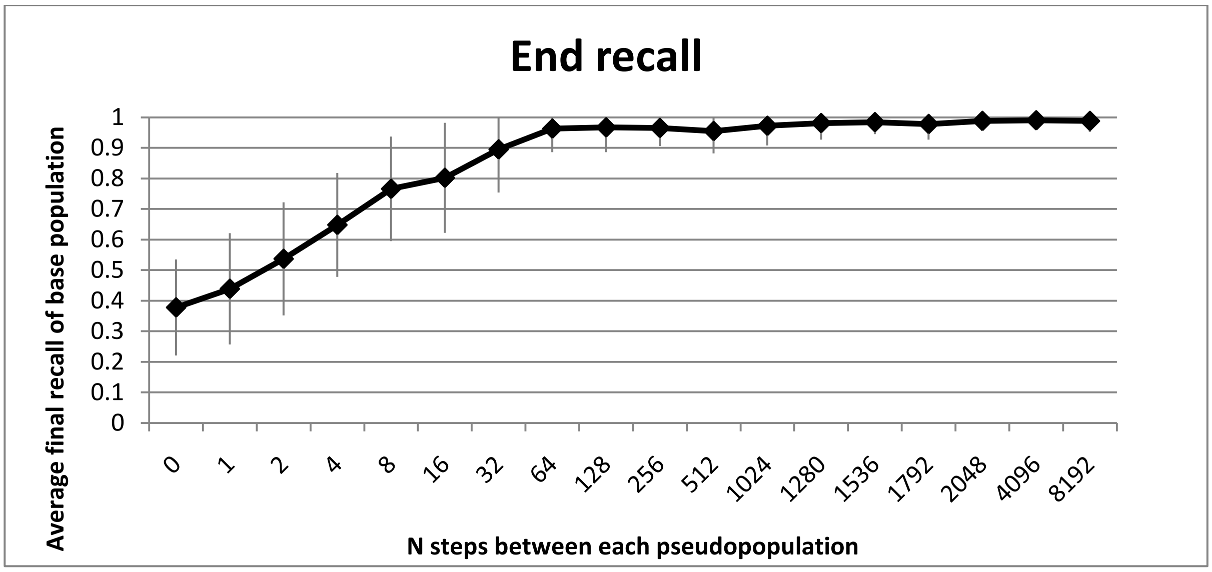


Figure 26: the same as Figure 22, but for larger values of N

4.6 Making pseudorehearsal more efficient

The figures above show that pseudorehearsal can effectively prevent catastrophic forgetting, but they also show that too much of it used in the wrong way is time consuming. This may not be important for all agents (in embodied agents for example, the number of steps is often more important than computing time), but it is still an issue in general. Given this problem, is there a way we can find a balance between minimal computational effort and maximum continual learning performance?

4.6.1 Using fewer pseudoitems

For agents using an infrequently generated pseudo-population, effort could be avoided either by generating fewer pseudoitems or by rehearsing with a smaller active buffer[14]. The experiments in this section focus on the 'new buffer each map' type of pseudorehearsal agent, because we have shown the other types to be either theoretically or in practice unsuitable for continual learning:

- The 'once only' agent has excellent continual learning performance, but is not generally useful if it can never permanently learn anything new
- The 'each step' agent has terrible continual learning performance, even though it has the theoretical advantage of not needing a persistent pseudoitem store
- The 'N steps' agent works reasonably well at some values of N, but is less reliable than the 'each map' agent because the data is less consistent (and a good value for N is hard to pin down)

[14] The "active buffer" is the set of pseudoitems scheduled to be rehearsed along with the next real input. Where the active buffer size is smaller than the pseudo-population size, it is good practice to cycle the active buffer through the pseudo-population (as in Robins' sweep pseudorehearsal).

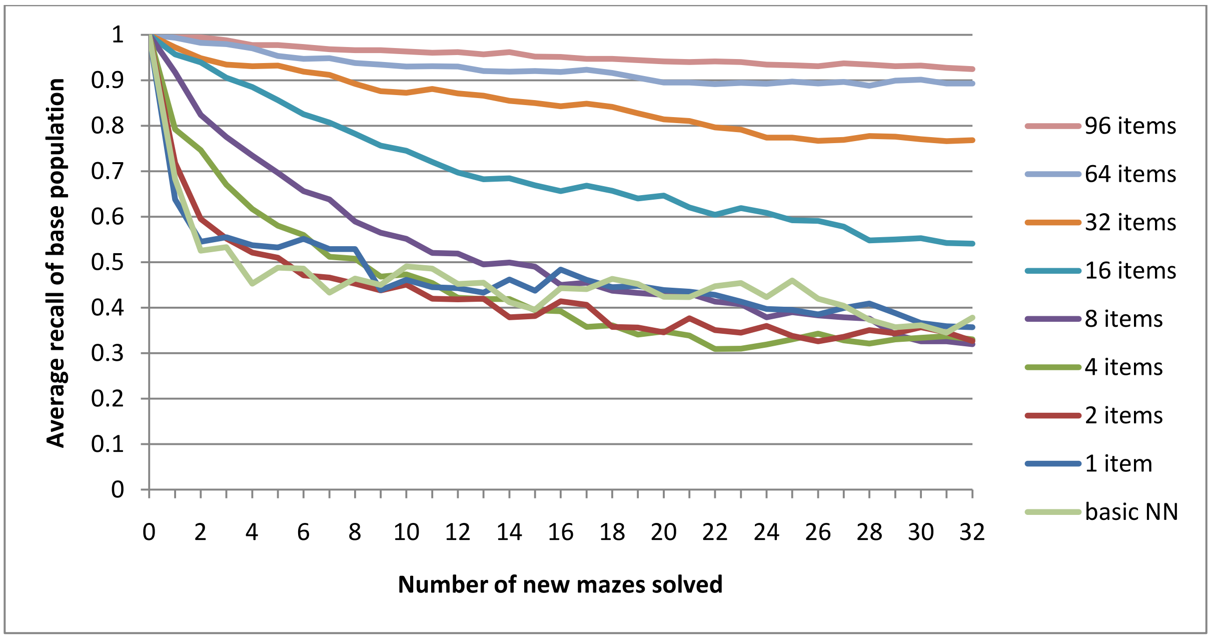


Figure 27: Performance of an agent using one new pseudo-population each map, with different pseudo-population sizes

Figure 27 shows the result of solving the same continual learning task with several different size pseudo-populations. These results show that without a medium- to large-sized pseudo-population, performance is not significantly improved relative to an unmodified neural network. At least 16 pseudoitems must be rehearsed along with each real item to give the agent noticeably better recall, and for nearly perfect recall (~90%) the agent must use a pseudo-population of 64 items.

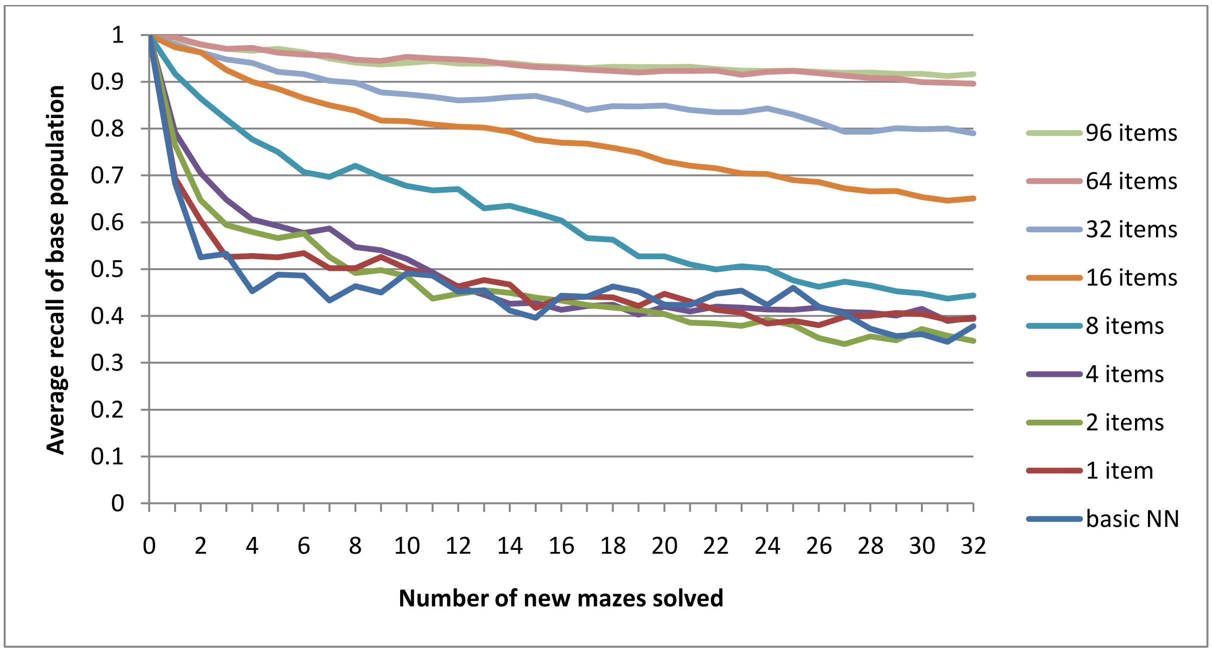


Figure 28: Performance of an agent using a new pseudo-population each map, with different active pseudo-buffer sizes

Instead of generating fewer pseudoitems to reduce computational overhead, another possible approach is to separate the pseudoitem population and the buffer used to actively rehearse pseudoitems. Figure 28 shows the result of solving the same task with an agent using a full-sized pseudo-population regenerated every time the agent finishes learning a new item, but with an active buffer of varying sizes. Performance is slightly better, but overall they are not significantly different. For comparison, the final recall of each agent for each of the different sizes is shown in Figure 29.

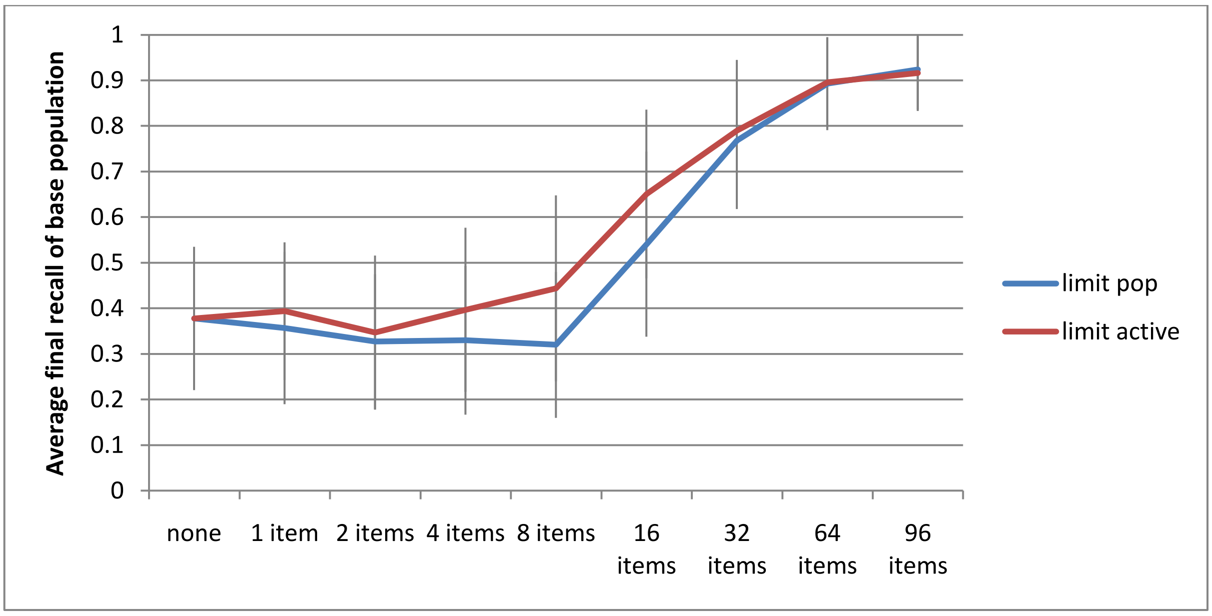


Figure 29: Comparison of Figures 27 and 28

Just as the continual learning performance of both strategies is roughly the same for both strategies, they also require a similar number of steps and CPU time to complete a trial.

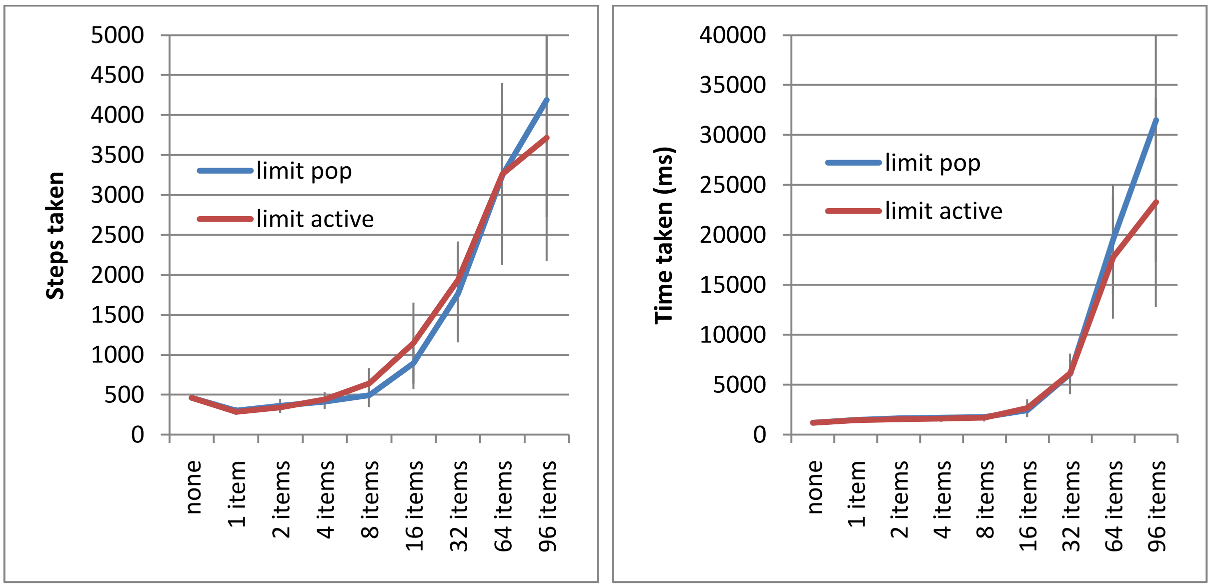


Figure 30: Steps and CPU time taken for both strategies

4.6.2 Using a different input representation

Section 4.6.1 showed us that, for this problem, partial pseudorehearsal was not really sufficient to avoid catastrophic forgetting, and the manner in which it was carried out had no effect on its efficacy. However, Section 4.5.3 showed us that full pseudorehearsal was not really a good option either, because it has a major detrimental effect on general learning performance (despite making continual learning possible).

The main contributor to this apparent failure may be the binary input representation used for this problem. Generally, it is very difficult for a neural network to learn associations between large sets of arbitrary binary inputs, because the data has a high degree of overlap. It is much easier for a neural network to learn mappings between inputs and outputs using a unary encoding, because the input mappings are orthogonal (i.e., there is no overlap between them). For our research, the term 'unary encoding' means that the network has one unit for each possible input, so if there were 32 states we would have 32 input units, and the state with number 17 would be represented as 00000000000000001000000000000000. For a binary encoding of the same range we would need 6 units and 17 would be encoded as 010001.

Unfortunately, it is often impossible to use a unary encoding in a machine learning problem. If the states in the sequential maze problem of section 4.2.1 were numbered using random 32-bit integers (instead of 7-bit sequential binary integers), then a network using a unary input encoding would need millions of input units, instead of just 32 for a binary encoding. On the other hand, if you know that there are not too many distinct states in a problem, it may be possible to avoid needing a binary representation by 'hashing' each distinct state to a unique integer small enough to use a unary encoding.

In this problem there are only 96 sequentially-numbered unique states[15] anyway, and it is not too infeasible to use 96 inputs instead of 8 or 32. This section examines the same problem, but with a unary input encoding for states.

The results depicted in Figure 31 show a comparison between an agent using a binary input representation and one using a unary representation. The unary agent has better continual learning performance, although it is still quite poor in absolute terms. The greater difference, however, is in the speed of learning by the two agents.

[15] Counting the 64 states in the large 'base population' maze and each new state in the 32 mini-mazes

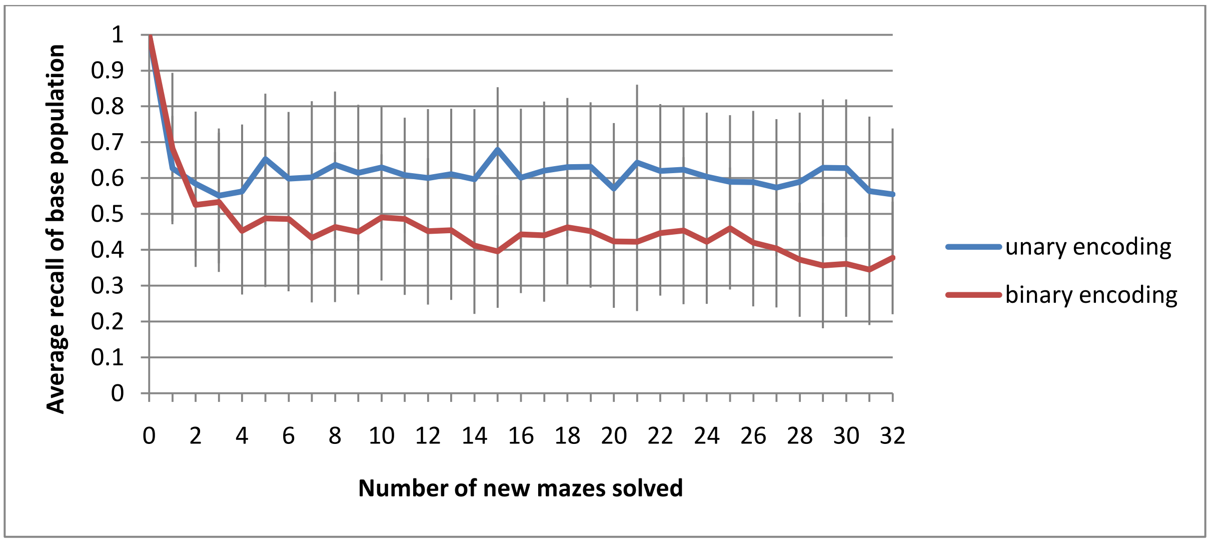


Figure 31: Recall performance for basic neural network agents using a distributed or localist encoding (i.e., a binary or unary input representation)

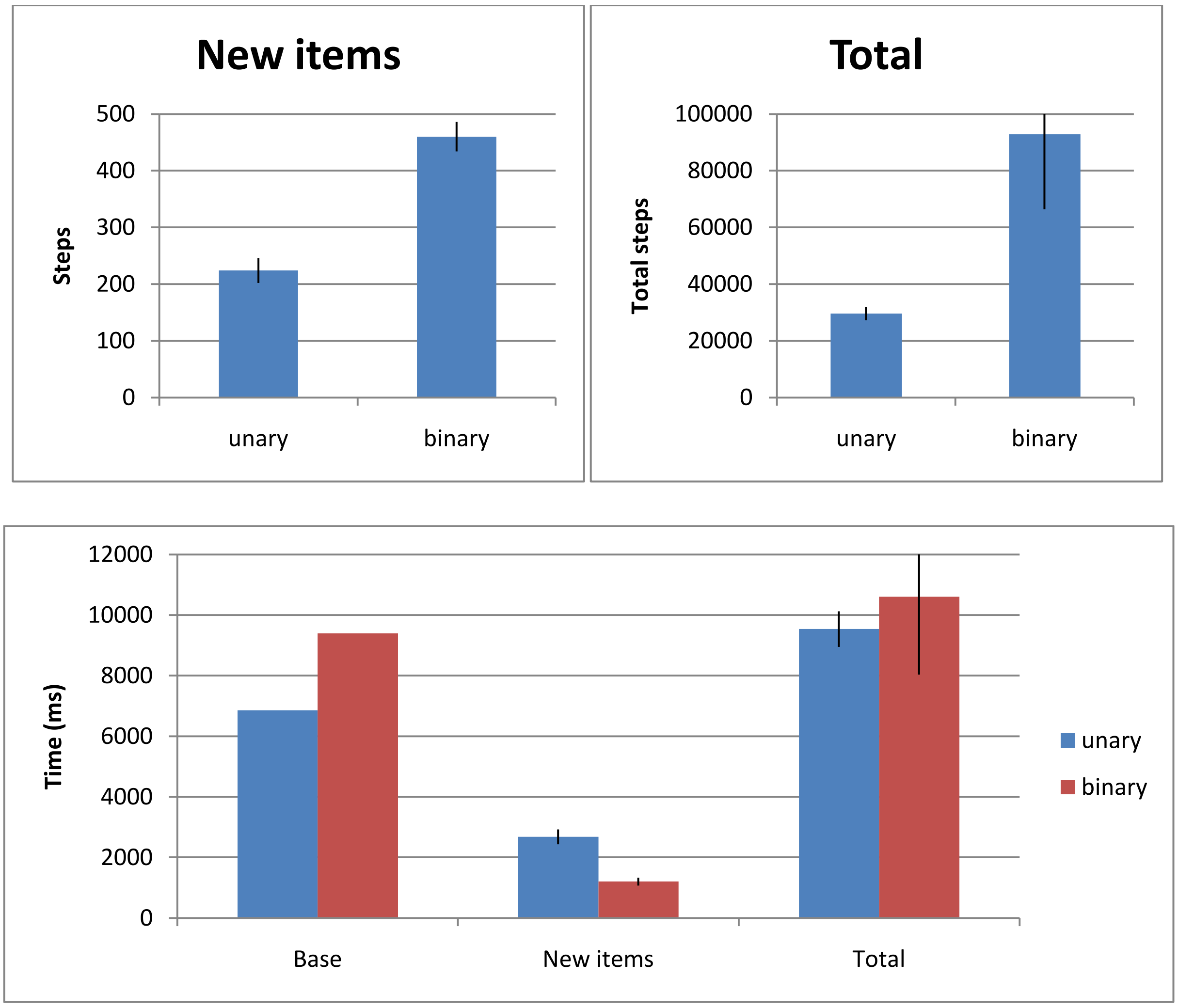


Figure 32: Times with various sized pseudo-populations for rehearsal, shown as a bar graph series

Figure 32 compares the performance of agents using unary and binary input representation. The agent using a binary input representation needs to take twice as many steps to learn the new items, and three times as many to learn the large base population. The cause of this is the same as the reason why the full-pseudorehearsal agents in Section 4.5.3 took so many more steps than the basic agent. Increased interference while learning decreases performance, combined with the fact that the network needs a more finely balanced weight configuration to learn non-orthogonal inputs than orthogonal ones. Due to the larger number of steps it needs to learn, the binary network also takes longer in milliseconds to complete the trials. However, the difference is largely mitigated by the increased computation per step with the unary agent (since the input layer has ten times as many units in the input layer, there are many more weights to process during learning).

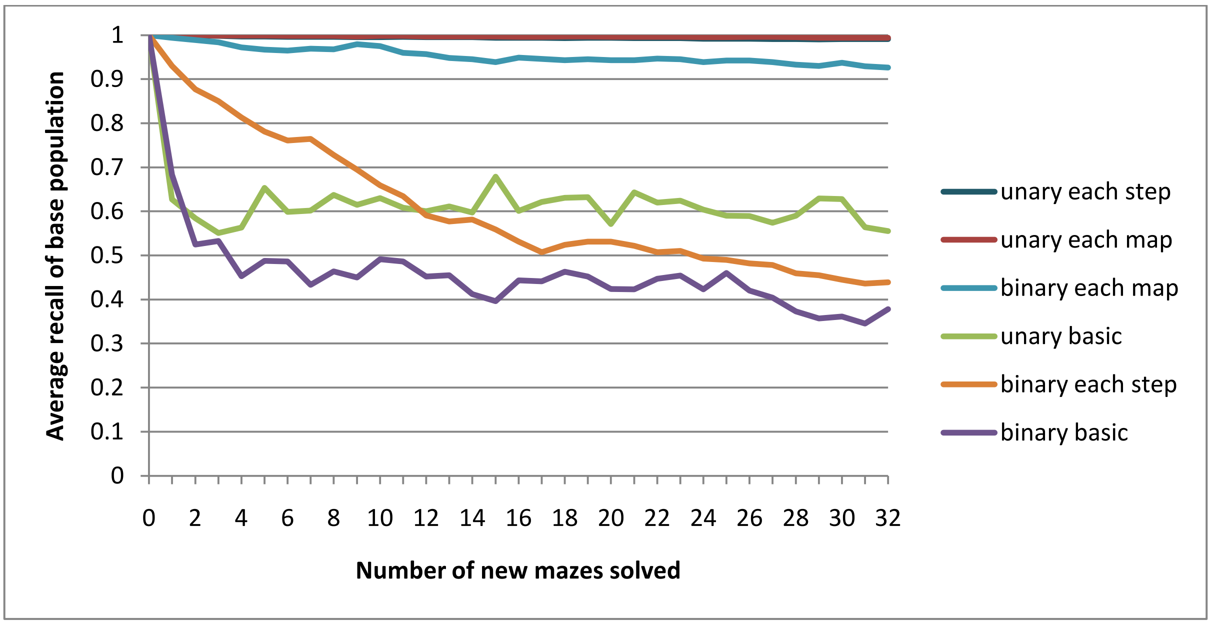


Figure 33: Comparison of recall performance with full-pseudorehearsal agents using a binary or unary input representation

While the difference between the unmodified neural nets when using a unary instead of binary encoding is significant, the difference between the full-pseudorehearsal agents is even more dramatic. Both the ‘each step’ and ‘each map’ agents have essentially perfect recall (>99%) at the end of each trial. CPU Performance is also improved, but less dramatically (Figure 34). The ‘each step’ agents take roughly the same time and steps to complete the trial (even though the unary agent has far better recall at the end), while the unary ‘each map’ agent is faster than the binary one (although they have similar recall performance).

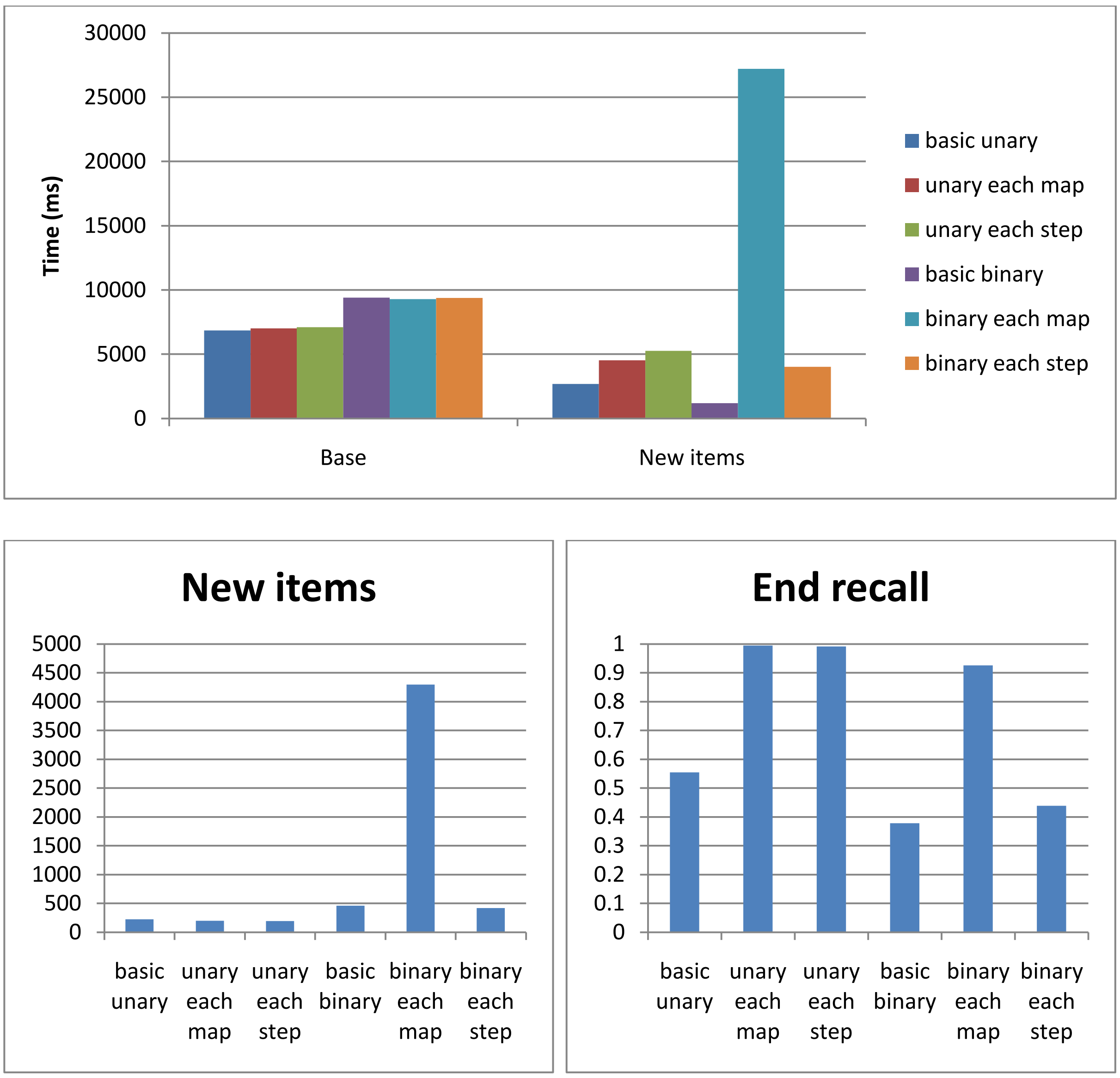
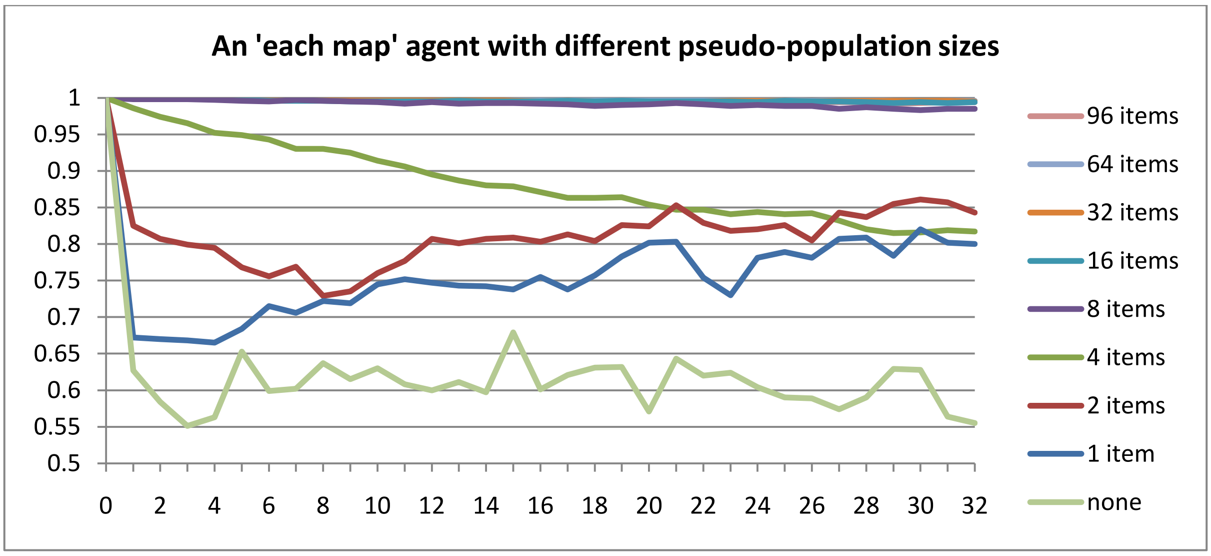


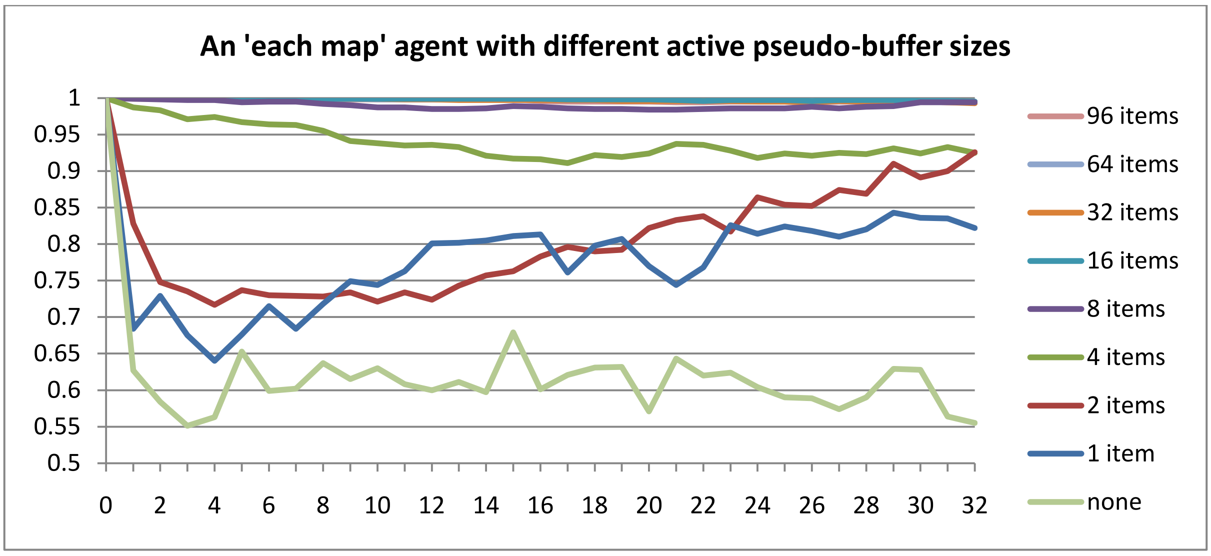
Figure 34: Comparison of performance with full-pseudorehearsal agents using a binary or unary input representation

Using a more orthogonal input encoding solves the problems of poor recall and poor performance for the full-pseudorehearsal agents. However, the unary agents are still slower (though less so) at learning than non-continual agents, and their perfect performance makes it clear that a lot of unnecessary rehearsal is being carried out. Figure 35 shows the results of reproducing the experiments of Figures 27 and 28 with a unary input encoding

4.6.3 A combination of techniques

Figure 35 shows the performance of agents using a variable-sized combined pseudo-population and active buffer (a), a separate full-sized pseudo-population and variable active buffer (b), and a pseudo-population regenerated every time the agent takes a step (c).





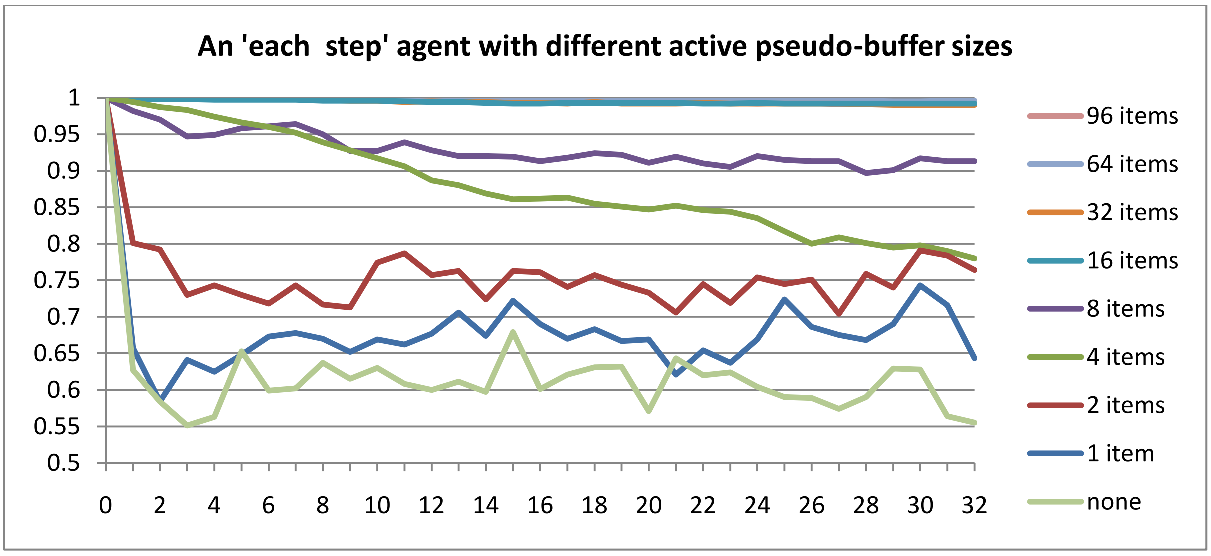


Figure 35: Performance of agents using variable-sized new pseudo-populations each map or each step, with a unary input encoding

Figures 25(a) and 35(b) match Figures 27 and 28, duplicating the results with a unary input representation. This time B is noticeably better than A, and B is twice as good as C (in terms of how many pseudoitems must be rehearsed to achieve a given performance level).

Figure 36 shows the time the agents take to learn the new maps (i.e., not counting the base map) in milliseconds (a) or total steps (b), and a summary of Figure 35 (c). The differences shown in Figure 35 are notable, but they are not nearly as dramatic as the differences in performance seen with the binary-coded input scheme. The reason for this is in Figure 36(b): every variation of the agent takes around 200 steps to learn the 32 new items, or around 6 steps per item. Generating the pseudo-population every 6 steps instead of every single step improves performance, but the difference is not extreme. Similarly, Figure 36(b) is better than 36(a), but the difference is only that in 36(b) the pseudoitems are cycled out every step instead of every 6 steps, on average.

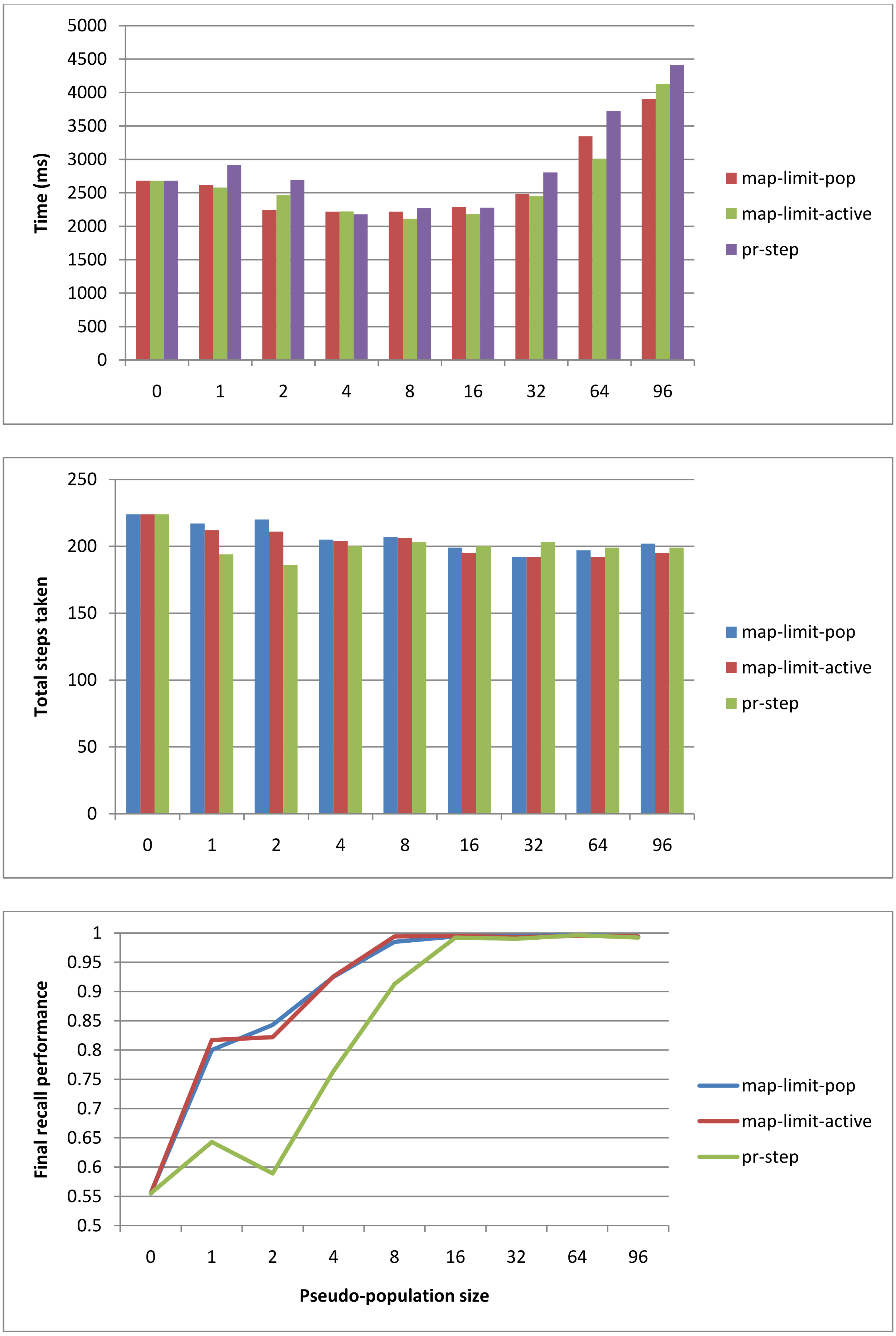


Figure 36: Performance of an agent using a new pseudo-population each step and a unary input encoding

4.6.4 Summary

The investigation in this section shows that pseudorehearsal can be used to overcome catastrophic forgetting in reinforcement learning problems when combined with a Q-learning agent.

However, the results also show that the technique can cause serious performance problems in certain situations. Neural networks are generally highly sensitive to the representation used to encode the current item (in this case, the current state) in the network's input layer, and using a different representation for the same value (for example, unary instead of binary) can have very significant effects. Pseudorehearsal magnifies this effect considerably (at least in this problem), by causing interference between real- and pseudo-items, which the network then has to overcome. Because of this the input representation should be as sparse as possible, and ideally the patterns are completely orthogonal.

When a sufficiently sparse representation is possible, pseudorehearsal can allow a neural network agent to learn many new items sequentially, without significant catastrophic forgetting and without an unreasonable performance penalty. However, the size of that penalty is problem-specific, and there are trade-offs between speed, continual learning performance, and storage outside the network to consider. In general, it is best to use pseudorehearsal with a large pseudo-population and a small active buffer of pseudoitems, and to replace the population at an interval that will stop new items being forgotten. We can also avoid the need to keep a separate store of pseudoitems by generating them instantaneously at every step, but this decreases performance.

4.7 Does pseudorehearsal speed up learning?

Baddeley (2008) found that a pseudopattern rehearsal strategy could speed up learning in certain types of (non-continual) reinforcement learning tasks. In our continual learning tasks we found that pseudorehearsal generally slowed things down instead. However, we would not necessarily expect pseudorehearsal to speed learning in those cases, since the new states were learned entirely independently of the network's existing knowledge.

If we restrict the problem used in Section 5.2 to just the base maze, and enable pseudorehearsal during learning, we can attempt to reproduce the trend Baddeley found in his results (but with a different type of problem and learning algorithm, of course). The neural network agents in this section used a learning constant of 0.3, momentum of 0.5, temperature set to 0.0, a unary input encoding, 32 hidden units, and the experiment was repeated 50 times and averaged to make the results more representative.

As in Section 4.5.1, training is stopped when the agent passes a test by finding its way to the goal in less than the maximum number of steps from every position in the maze. Although accuracy could be quantified during learning by measuring the proportion of successful test episodes, the average number of steps per test trial, or some combination thereof, the figures below only show total training time. Baddeley's (2008) main thesis was that total training speed was reduced due to catastrophic forgetting, and besides plotting the 'goodness' during training of a few different agent configurations, he made no attempt to quantify whether the 'rate of goodness increase' was improved with his pseudopattern strategy, only total training time.

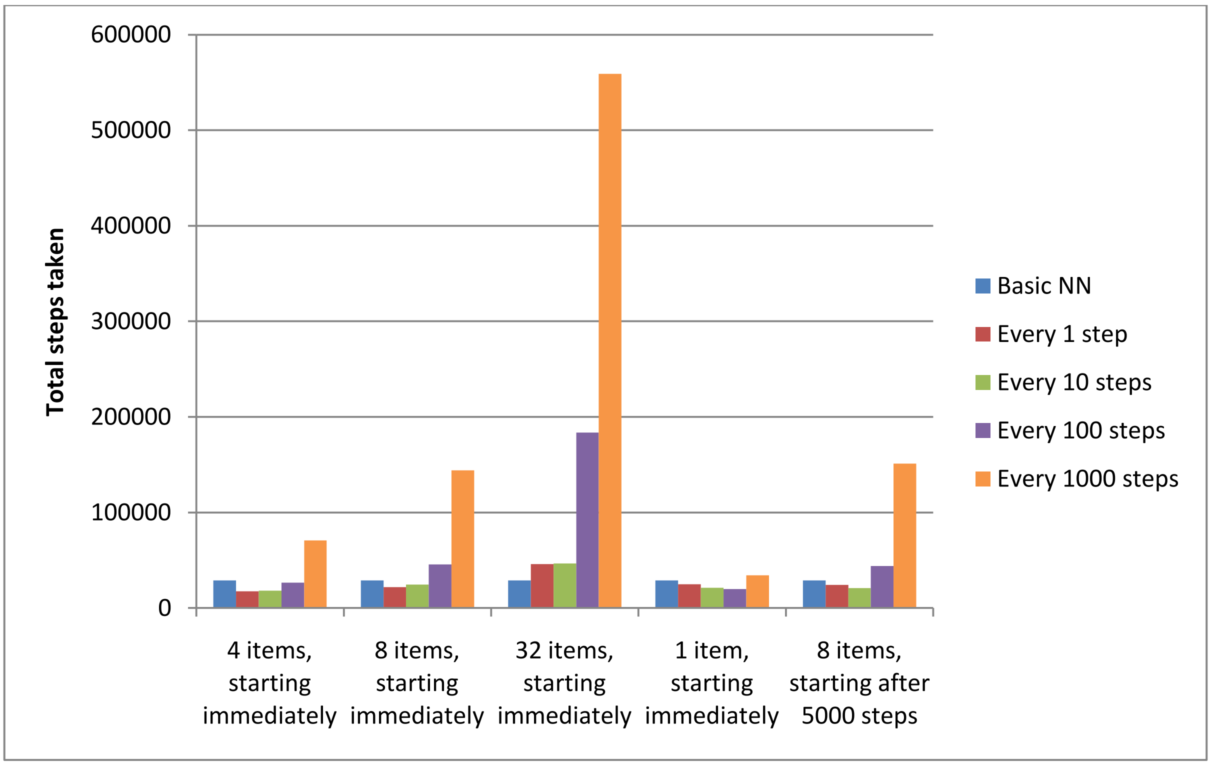


Figure 37: The number of steps taken to learn a single maze from scratch, using a variety of pseudorehearsal configurations

Figure 37 shows the number of steps taken during training by agents learning just the single large base maze, with a number of different pseudorehearsal configurations. The results show that for at least some configurations, pseudorehearsal provides a significant speed benefit. However, other settings can cause severe performance problems, and for some, the agent may completely fail to converge (meaning the bar would be infinite).

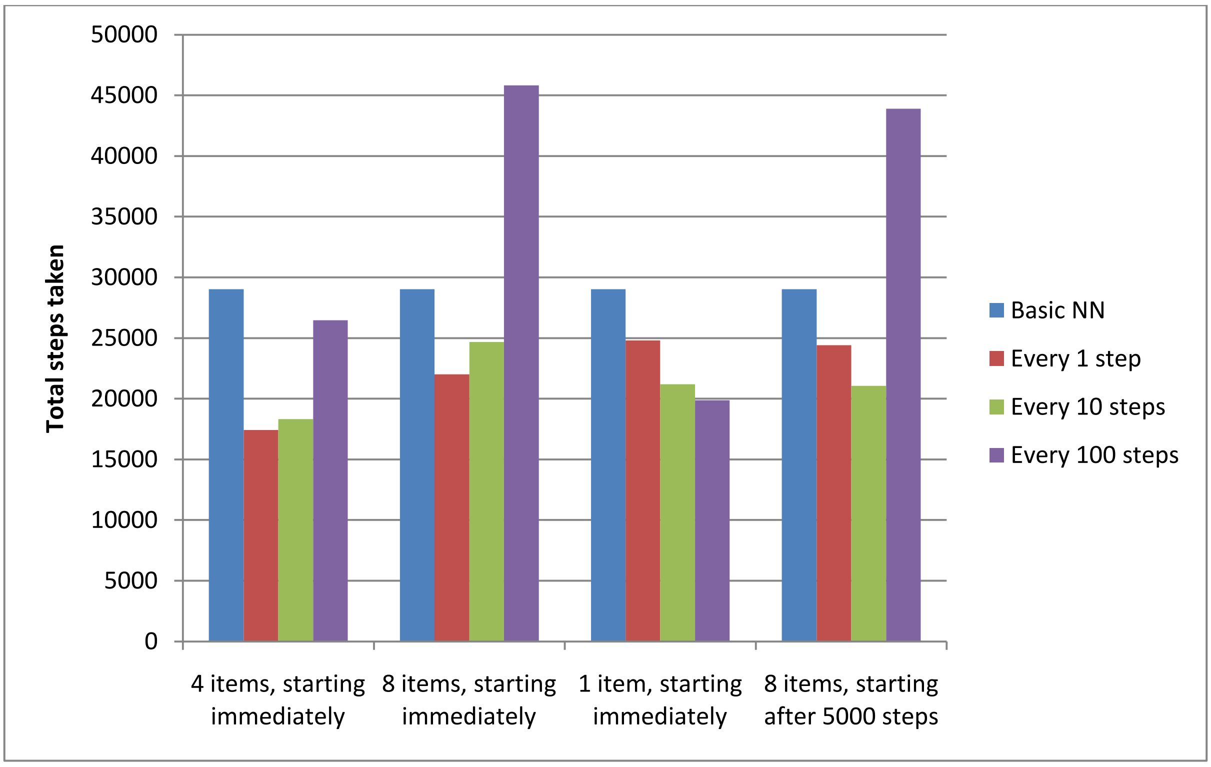


Figure 38: The same as Figure 35, with only the more effective configurations shown

Figure 38 shows the more effective configurations for using pseudorehearsal during initial (i.e., non-continual) learning. Small numbers of actively rehearsed pseudoitems regularly regenerated works best, and one of the pseudorehearsal agents was 40% faster than the standard agent.

On the other hand, the same agent took 2.5 times longer than the basic agent to complete the training (Figure 39). The best all-round agent using 1 active item regenerate every 100 steps took only slightly more steps than the most efficient agent, but it still took slightly longer than the basic agent to finish the trial.

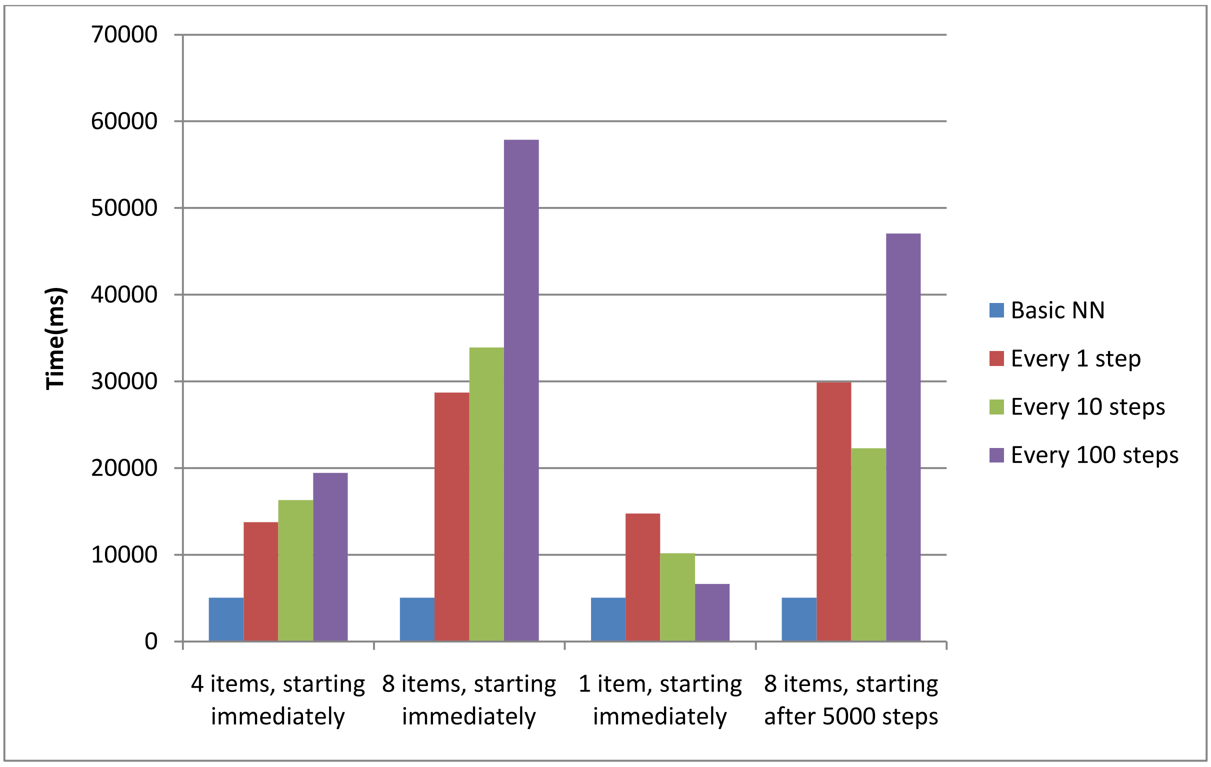


Figure 39: Time taken in milliseconds by the agents in Figure 38

4.7.1 Summary

The results in this section show that pseudorehearsal agents can in principle learn faster than standard neural network RL agents, when measured in the number of steps they needed to take before they were able to solve the maze perfectly, and they can be up to 40% faster. This is in line with the results from Baddeley (2008), which found a dramatic increase in performance. However, the improvement was far less drastic than Baddeley's, which was greater than 90%.

In contrast to Baddeley (2008), the agents were still generally slower than their non-continual counterpart in thread CPU time needed, due to the additional computation pseudorehearsal requires. On the other hand, the agents in this trial were not optimised in the same way as Baddeley's were, and the learning algorithms were not identical. It is likely careful parameter optimisation could reduce the performance gap further.

4.8 Relearning effect investigation

The review in Robins (2004) pointed out that context biasing agents are unable to retain information learned sequentially well enough to recall it adequately without retraining, so they do not perform well on supervised continual learning experiments testing exact recall. However, the results in Section 5.2 showed that orthogonalising the data in the input layer of the network improved the continual-learning performance of the basic neural network RL agent. In a similar manner, Context Biasing works by orthogonalising the data in the hidden layer of a neural network. It is therefore not unreasonable to expect the use of Context Biasing to have a positive effect on continual-learning performance compared to a standard agent. However, the continual learning performance of a context biasing agent may still be worse than a pseudorehearsal agent.

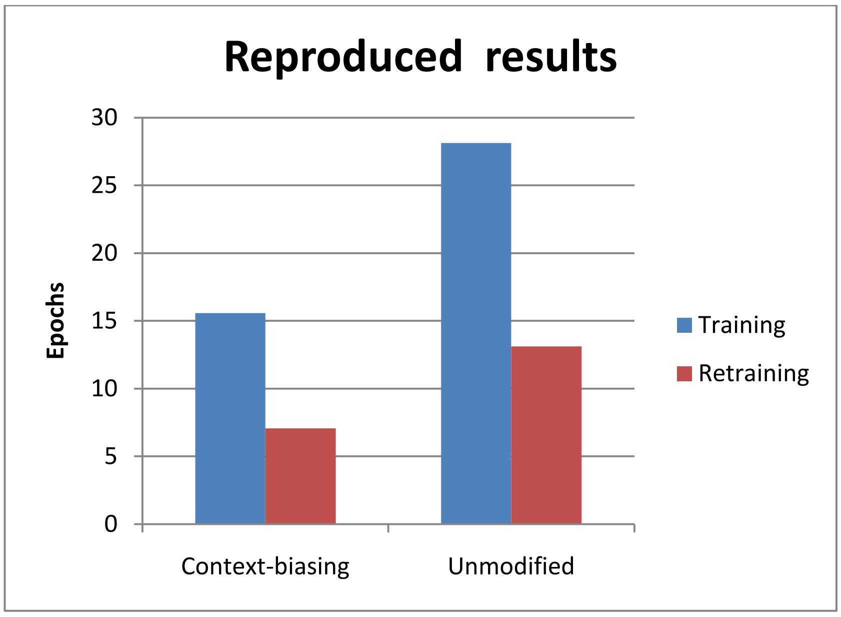
4.8.1 Replicating French's results

Before testing the Context Biasing algorithm in a reinforcement-learning context, we must implement it and attempt to replicate the main result from French (1994). The purpose of this experiment is therefore to give us some assurance that the basic supervised implementation is correct.

For his experiments, French used a subset of the 1984 US Congressional Voting Records dataset. He did not state exactly what subset, or exactly how he generated his secondary dataset, so a reasonable approximation of his dataset was used (included in Appendix A).

French trained a 16-10-1 feedforward backpropagation network to associate 50 different voting patterns with party affiliation. He then invented a small set of ten "maverick" members of Congress, members who, on six key issues, voted like Democrats but declared themselves to be Republicans or vice-versa. He had the network learn this new set and then re-tested its performance on the original set of fifty associations, which he found was completely forgotten. Following this, he retrained the network on the original dataset. French called the speed with which the network relearned the original data a “measure of savings”, showing how completely the network had forgotten the original data. This “relearning effect” is discussed in more detail in French (1999).

Figure 40 shows the result of replicating French’s experiment. As expected, the results are nearly identical.



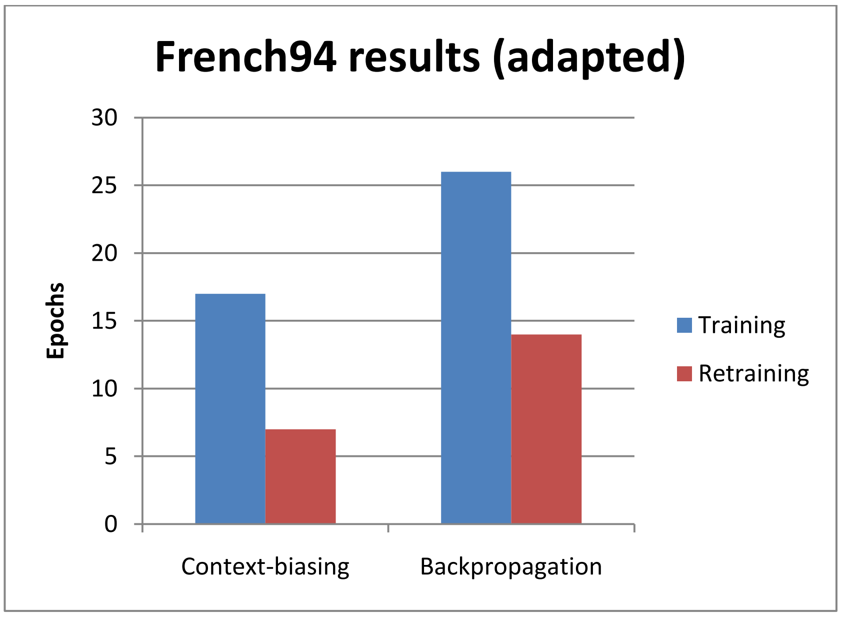


Figure 40: The results (above) from replicating the experiment in French (1994), with the original result for comparison (below)

4.8.2 Continual learning experiment

As mentioned by French (1999), one interesting property of neural networks is that although they may easily forget a base population while learning a new population, it is often the case that the same network can be re-taught the disrupted base population in less time than it took to learn that population originally. This experiment measures that property, called the relearning effect. The environment used in this experiment is identical to the last, only the method of experimentation is slightly changed:

For N from 1 to 20, repeat:

- Learn the base population maze
- Learn N new mini-mazes sequentially, in random order
- Measure and record how long it takes the agent to relearn the original maze

This experiment was also repeated 50 times, with the same process for training and testing the agent.

4.8.3 Continual learning performance

Despite the improvements shown in the supervised results above, the RL context biasing agent performs no better than the basic neural net agent, even with a range of different parameter settings. The best result, (shown in Figure 41) with β=0.5 and the other parameters as in Figure 40, is comparable to the basic agent. Apart from the additional parameter β, both results shown here were generated with agents using the same settings as in Figure 18.

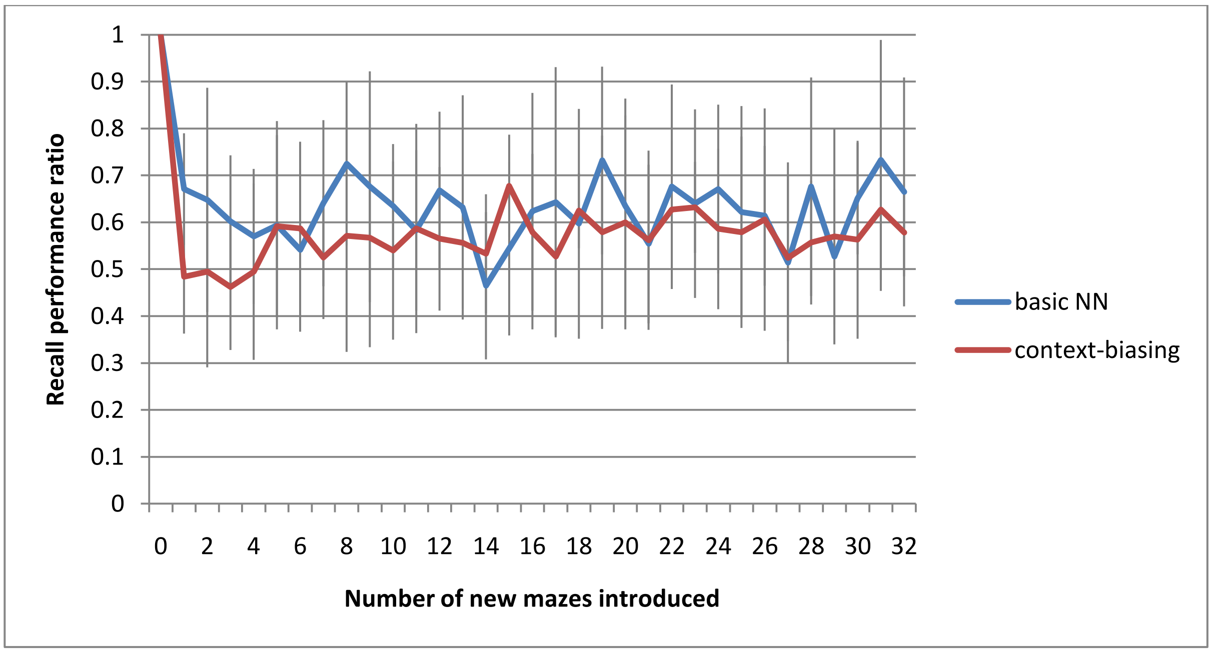


Figure 41: Comparison of performance between a basic neural net and a context biasing agent, for the continual learning experiment

4.8.4 Relearning

Since context biasing is generally poor at maintaining perfect recall, we attempted to gauge the ‘measure of savings’ French (1991) introduces by using the relearning experiment from Section 4.7.2. This experiment measures the amount of training needed (measured by recording the number of steps taken during training) for an RL agent to relearn a base population after learning N intervening items.

This experiment was carried out with a standard neural network agent, a context biasing agent (both using the same settings as before), and a pseudorehearsal agent (using full pseudorehearsal as in Figure 18, with an active buffer regenerated after each new item). Figure 42 shows the results of that experiment. The context biasing agent is modestly faster than the basic neural net agent, but compared to the pseudorehearsal agent, it has poor performance in this trial.

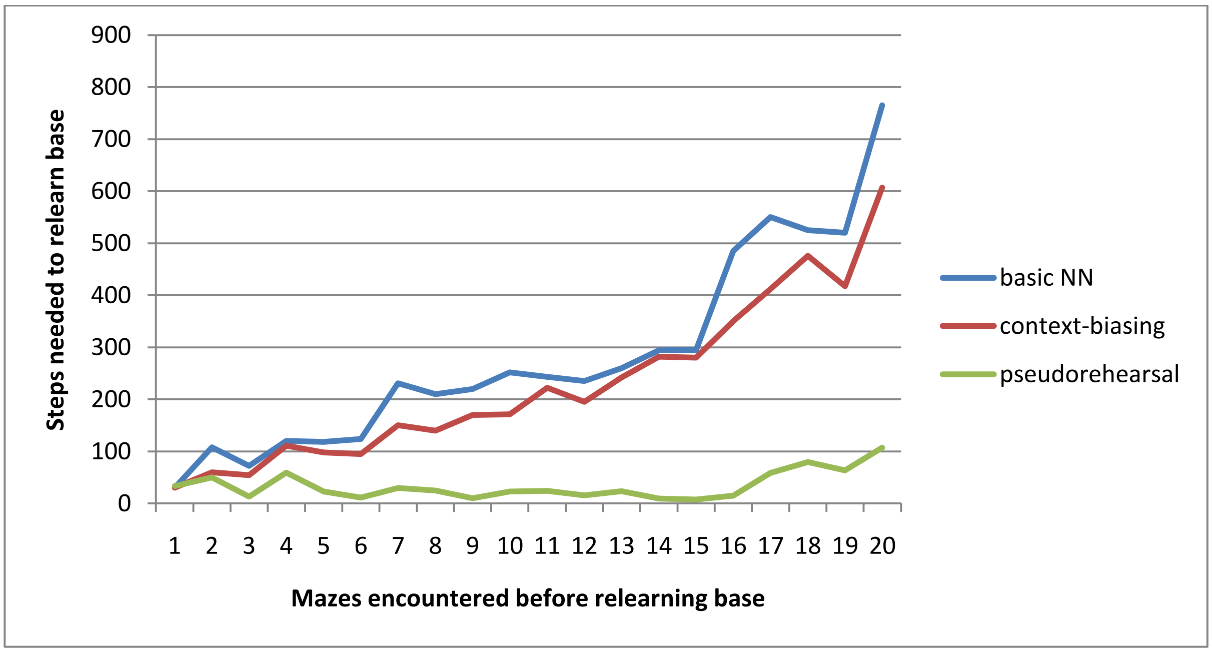


Figure 42: Comparison of agents completing the relearning experiment

4.8.5 Summary

In this section we successfully reproduced French’s Context Biasing neural network algorithm, and also incorporated it into a Q-learning RL agent. Despite the significantly improved performance in the supervised test (Figure 38), the RL agent is not greatly superior to a basic neural net RL agent, while the pseudorehearsal agent has far better recall and relearning performance than the other agents.

4.9 Continual learning POMDP experiment

One of the most important reasons why a continual learning agent is useful is that it can draw on its existing knowledge to learn new tasks quickly, just as intelligent biological agents can. The tests in this section quantify that ability using a version of the experimental maze design of Ring (1994).

4.9.1 Continual learning experiments

This experiment was essentially a duplication of the experimental setup Mark Ring used in section 7.3 of his thesis (Ring, 1994). Ring's experiments are a carefully designed test of continual learning (transfer of knowledge) performance, and it allows us to make a fair comparison to his Temporal Transition Hierarchies algorithm.

For each of the nine maps from Figure 16:

1. Train the agent through 100 trials of the map (stochastic action selection)
2. Test the agent on the map for 100 trials (deterministic actions)
3. If the agent reached the goal on every trial of the test, go to the next map, otherwise go back to 1.

The test was repeated as with the others, to get a more representative measurement for each agent. Two versions of this experiment were run. In the first, the agent was completely reset at the start of each new map it encountered. In the second, the agent was allowed to keep its existing knowledge. Comparing these results allows us to determine whether the continual learning agents are able to successfully transfer their knowledge to new tasks, by measuring training times relative to the non-continual version of the experiment.

4.9.2 Agent configuration and parameters

The representation of each state in this problem is derived solely from the presence or absence of four walls around the agent, so it would be possible to use four binary units to encode the current state in the network's input layer. This has the theoretical advantage of allowing the network to recognise similarities between distinct states based on the presence or absence of specific surrounding obstacles, possibly increasing learning and generalisation. However, the investigation in Section 4.5.2 shows that the neural network agents are likely to be too sensitive to interference between closely overlapping units to take advantage of this, and the continual learning results in Ring (1994) also use a unary input representation. To avoid unfairly handicapping any of the agents in this test, all of them used unary representation for the experiments in this section, and the TTH agent used exactly the same parameters as Ring (1994).

4.9.3 Results

Figure 43 shows the performance of an agent using Ring's Temporal Transition Hierarchies to solve each maze in Ring's continual learning task from scratch. Also shown is a Q-learning agent using a simple lookup table[16] and delay lines (introduced in Section 3.7.1) to allow it to "remember" the three previously visited states. The tabular agent used a learning constant of 0.3, temperature constant of 2.1, and a discount factor of 0.9.

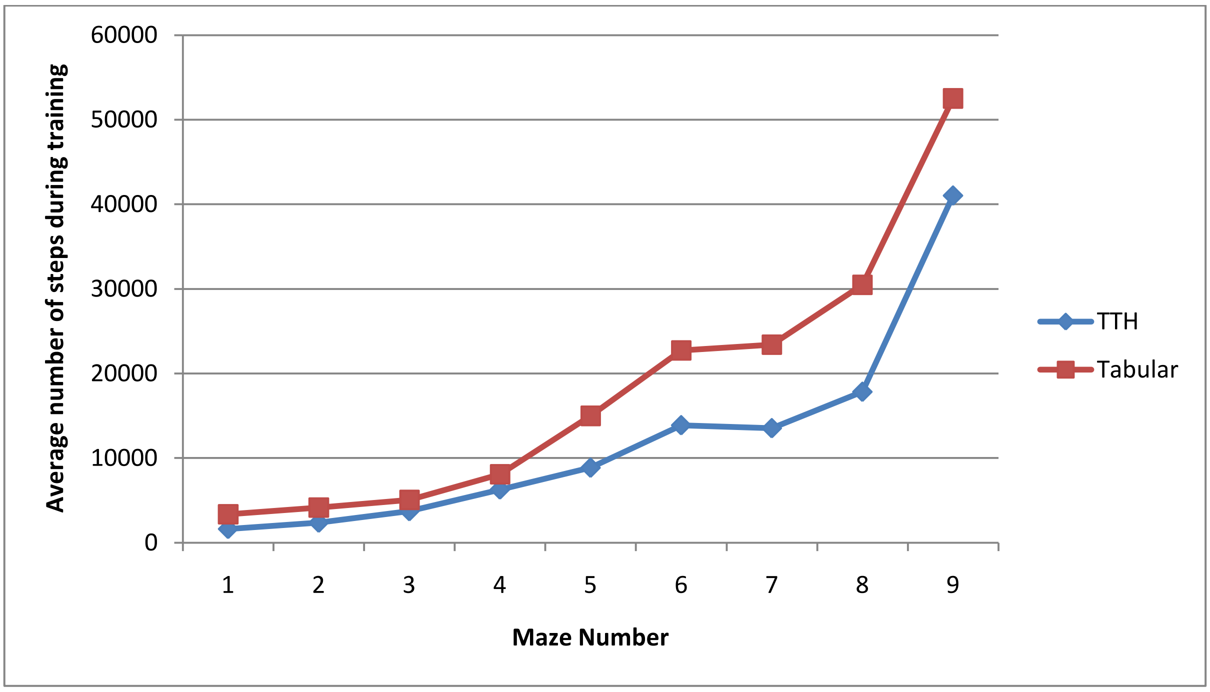


Figure 43: Performance of a CHILD and a Tabular agent learning each maze in Ring's continual learning problem from scratch

The results suggest that delay lines are an adequate tool for use in POMDPs, but they may be less efficient than the constructive perceptual window effect of TTH, which learns to optimise the exact number of states to recall. Alternatively, it may simply be the case that TTH converges significantly faster than a Tabular algorithm.

[16] Actually, the agent's internal representation is a hash table keyed by state histories (integer arrays).

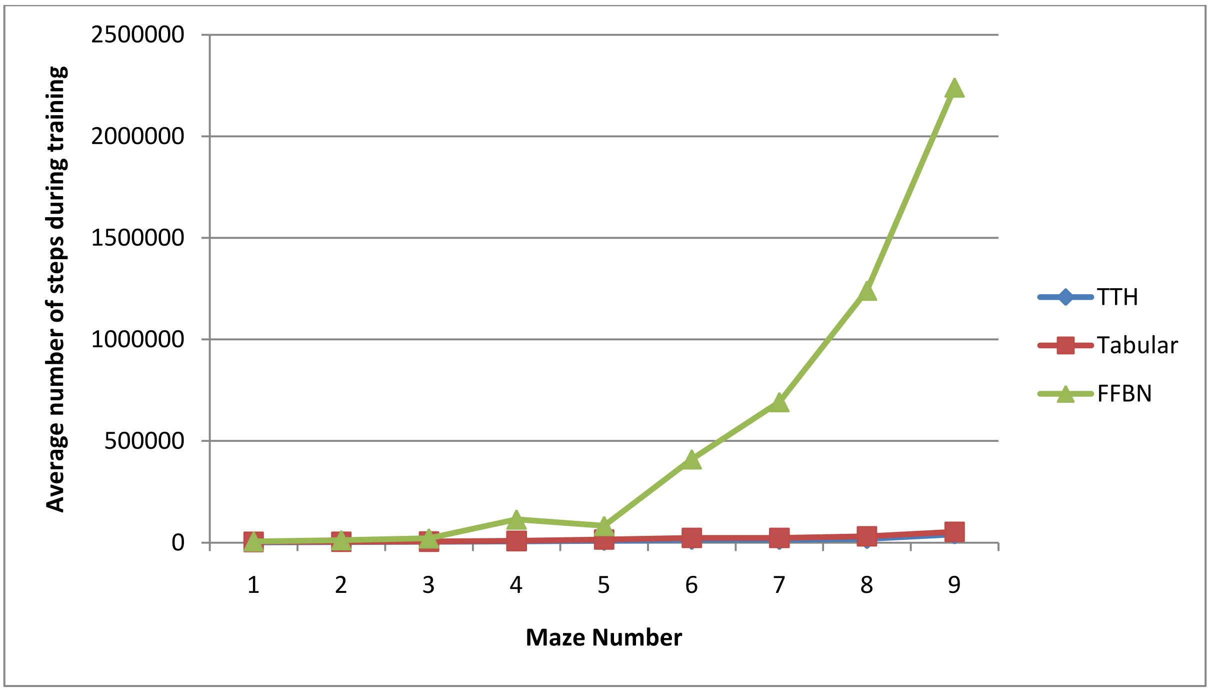


Figure 44: The same as Figure 43, but with a rescaled vertical axis and including a basic neural network Q-learning agent

Figure 44 shows the performance of a basic neural network Q-learning agent with parameters optimised for the maze environments. It used 64 hidden units, a learning constant of 0.3, temperature constant of 2.1, and a discount factor of 0.9, and a delay line window of 3 (the same as the tabular agent). The agent has comparable performance to the others for the first five mazes, but it is almost incapable of solving the more difficult tasks, taking around 100 times longer on the last maze. However, the continual learning case (Figure 45) is very different to the picture shown by Figure 44.

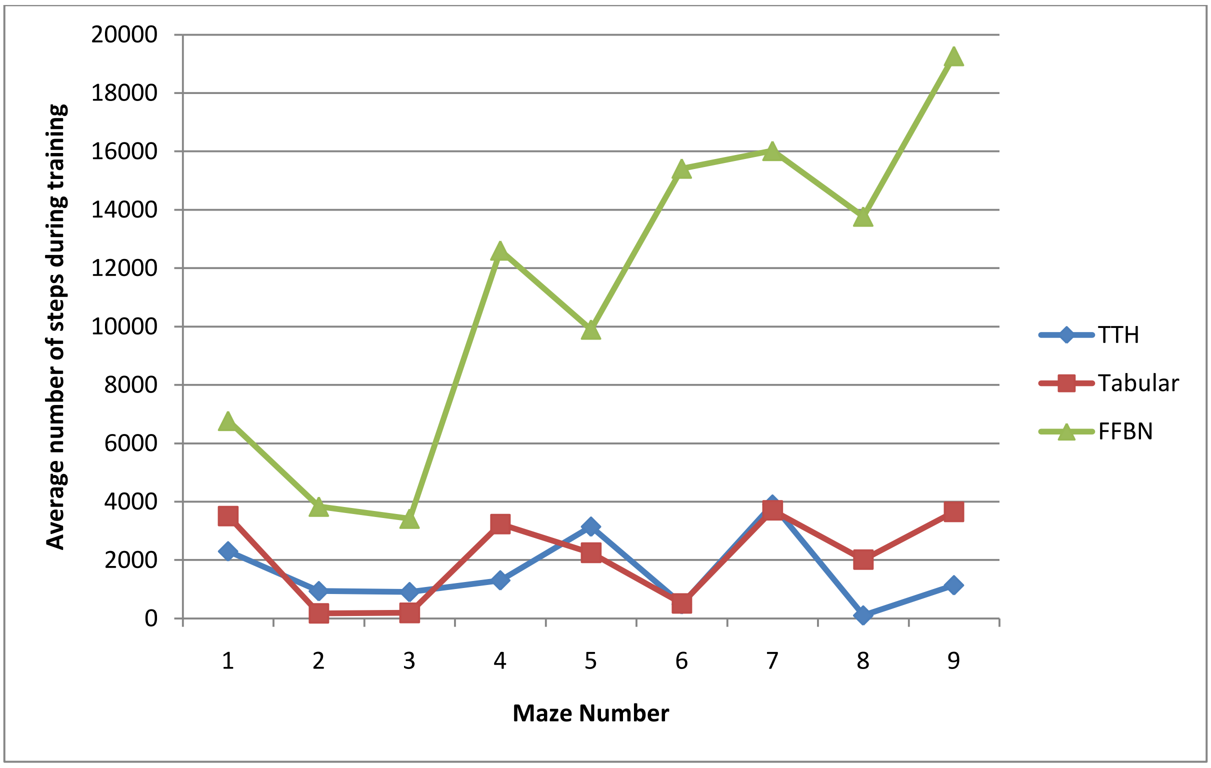


Figure 45: The same as Figure 44, but the agents are not reset after solving each maze

Not surprisingly, the tabular agent performs very well on the continual version of Ring's experiments, with results essentially identical to CHILD. The performance of the neural network agent is much closer to the other agents in this case as well, with performance only being 4 to 5 times worse, instead of 100 times slower. This is somewhat surprising as Ring (1994) concluded that CHILD was an effective continual learning agent on the basis that it is able to learn the mazes more quickly sequentially rather than learning each one from scratch. However, as we demonstrated in Section 5.1, an unmodified neural network RL agent has very poor sequential learning ability. Instead, the improvement appears to be the result of the phenomenon of *shaping*, which Ring also describes. The earlier simple mazes act as a rough template for the difficult mazes, and it is much easier for the network to fill in the details that way than to solve the difficult problems without a guide. This effect was also documented in detail by Rountree (2007), which pioneered the use of decision trees as a rough approximation to initialise a neural network (for supervised problems).

Even so, the basic neural network agent has poor performance compared to the TTH agent, and this may (at least in part) be due to catastrophic forgetting. If so, then pseudorehearsal could be useful for improving the performance of the neural network agent. Full pseudorehearsal (generating a pseudoitem for every visited state) is certainly not possible in this case because for that the agent would need to rehearse 65,536 (16^4) pseudoitems for each real item[17]. Figure 46 shows the relative performance of the more useful pseudorehearsal configurations, with the pseudoitem population refreshed after each map, after every step, and every 100 steps (N=100 was found to be the most effective setting by trial and error, in this case). The pseudorehearsal agents used an active buffer of 8 items. A context biasing agent (β=0.5 had the best performance) is also included.

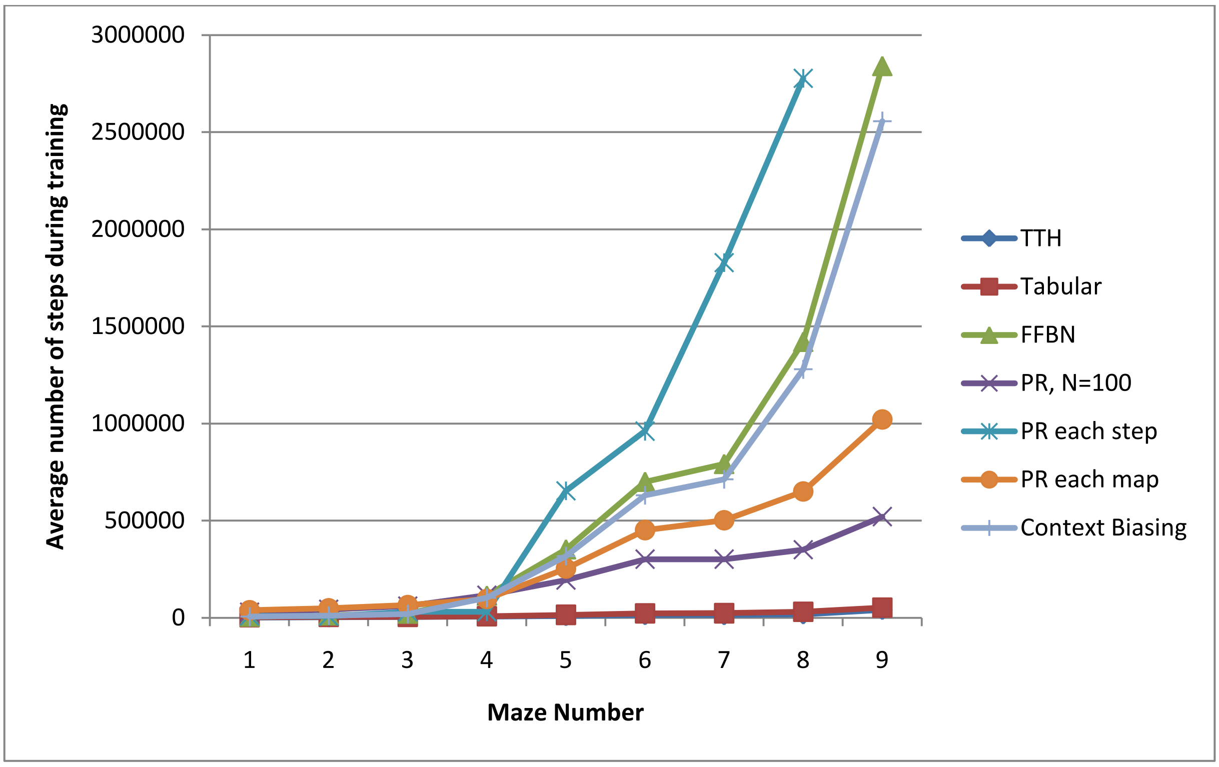


Figure 46: The same as Figure 44, but with three differently configured pseudorehearsal RL agents added

The different pseudorehearsal schemes have varying performance, and surprisingly the worst is the 'each step' agent, which is actually worse than the standard NN agent. Context Biasing is only slightly better than the unmodified agent. The best performer is the agent using a new pseudo-population every 100 steps. Even though it learns the difficult mazes more slowly than the TTH and tabular agents it is much closer (around 10 times) than the non-

[17] Assuming that the RL agent is configured so that the current input is presented to the network along with the three previous input states, there would be 16 possible inputs for each of the 4 states in the 'perception window', for a total of 16^4 states

pseudorehearsal neural network agents, and is reasonably close to the TTH agent, with less than an order of magnitude difference between them.

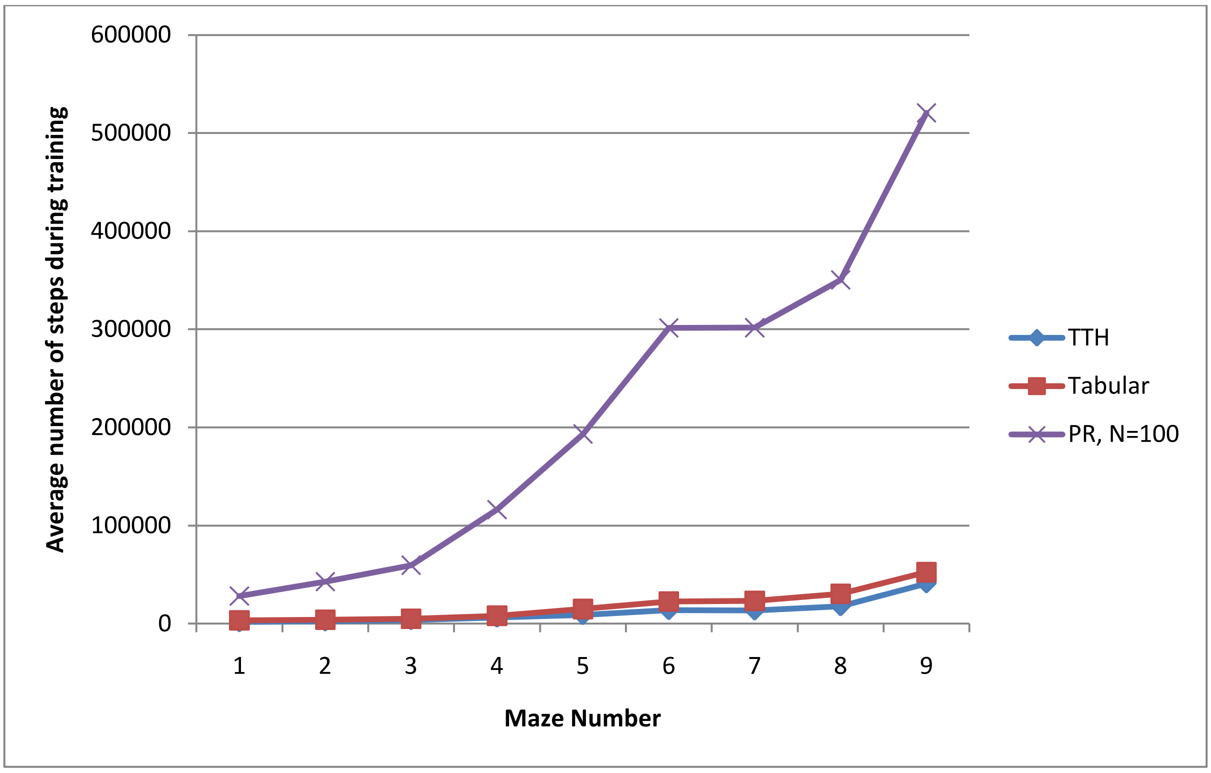


Figure 47: The best performing agents from Figure 46, with a rescaled vertical axis

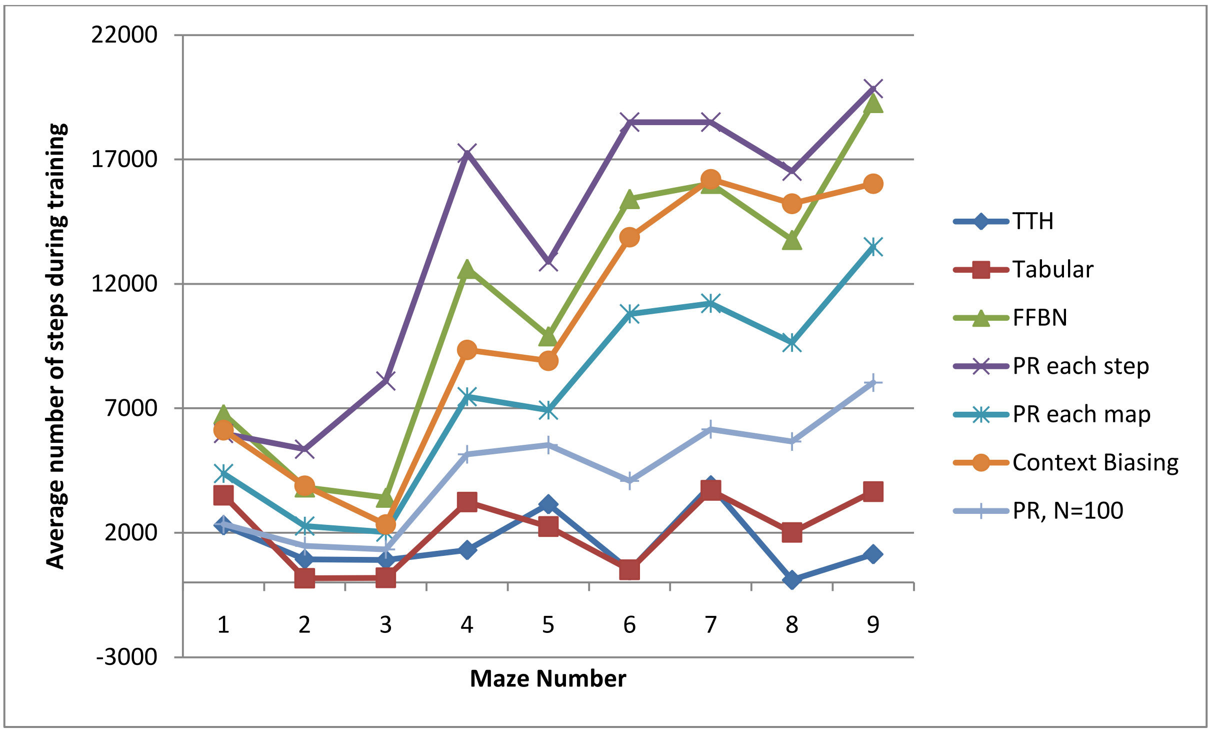


Figure 48: The same as Figure 45, but with pseudorehearsal and context biasing agents

In Ring's continual learning test (the same as Figure 45), the N=100 agent is also the best performing pseudorehearsal agent. Despite having less exhaustively optimised parameter settings, the agent is only around three times slower than CHILD (all of the pseudorehearsal agents used the same settings as in Figure 47).

4.9.4 Summary

The final stage of the investigation in this thesis was to directly compare a Q-learning neural network agent (using pseudorehearsal and delay lines) to other continual learning RL agents, with the POMDP in Ring (1994) as a benchmark. The pseudorehearsal agent was shown to be significantly slower than the much more complicated and finely tuned Temporal Transition Hierarchies agent. However, even without exhaustively optimising the pseudorehearsal agent's parameters, it is only around two to five times slower than TTH; close enough for it to be considered adequate, and it is certainly far simpler to use.

5. Discussion

The aim of this research was to investigate the types of problems that intelligent agents need to solve to operate in the real world. In particular, those agents need to efficiently solve problems involving only limited reinforcement-level feedback. These problems are more difficult in contrast to entirely supervised problems (with perfect feedback), which tend to be vanishingly rare outside of artificial settings. We also considered the issue of solving RL problems continually, throughout the life of the agent, without excessive inefficiency or interference. Additionally, we investigated solving lifelong RL problems with neural networks, which make for a promising avenue of research given the success of the source of their inspiration, the large body of existing research on them, and their well understood yet powerful learning capability.

Successful examples of continual learning neural networks (Robins, 2004) and RL agents that use neural networks (Tesauro, 1994) are not new. Individually, reinforcement learning, neural networks, and continual/lifelong learning are well-studied problems, but the combination of the disciplines is only beginning to be researched.

Although research on RL agents employing neural network systems to solve sequential or continual learning problems is relatively uncommon, there is some existing research in this area. In particular, Baddeley (2008) studied the application of the pseudopattern strategy to RL problems, and Ring (1994) developed a specialised type of neural network RL agent capable of efficiently solving certain types of continual RL problems. However, Baddeley did not investigate solving continual RL problems, and only observed that catastrophic forgetting appeared to be degrading the performance of an RL agent learning to solve a non-continual problem. Ring's system, CHILD, was very successful at solving the problems he designed to test it, but it used a complicated constructive temporal learning algorithm with an unusual linear network design that (from experience) is difficult to implement and use reliably. Other researchers (Provost, 2007; Mitchell, 2003) have noted that CHILD could be very useful for solving continual reinforcement learning problems, but there has not been much actual experimentation, possibly due to the difficulty of implementation.

Therefore, one important goal of this research was to investigate ways of solving continual RL problems that were simpler to use and benchmark against, but still powerful enough to solve interesting problems[18]. For this reason, the practical experimentation of this work was

[18] Note: a simple Q-learning agent with a lookup table utility store is perfectly capable of continual learning (since its entirely local representations are not subject to interference), but it lacks generalisation and does not

implemented in Java using the RL-Glue framework, a modular cross platform framework designed to encourage open research and foster academic collaboration in the machine learning community.

5.1 Results

The results in Section 4.4 showed that catastrophic forgetting is still a serious problem for RL agents using a neural network as a value function approximator, just as it is an issue for directly learning supervised mappings with a neural network. However, it appears to be the case that the forgetting in RL is more gradual, which is why a non-continual agent can still be reasonably successful despite the issue of conceptual drift (see Section 3.8)

One reason why CF is more gradual is because the goal of reinforcement learning is usually[19] different to the goal of supervised learning. Instead of perfectly reproducing a mapping between supervised inputs and outputs, the RL agent only has to generate predictions of future reward (DFR) that are accurate enough for it to behave acceptably well. So even if the optimal DFR vector for the actions available in a given state is something like $[0.83, 0.75, 1.0, 0.91]$, the agent could get away with learning $[0.99, 0.99, 1.0, 0.99]$ as long as its behaviour and the transition function of the environment are both deterministic. Because the agent needs to be less finely tuned, catastrophic forgetting may not arise so quickly. Therefore, one useful rule for reducing CF in RL problems is to stop training as early as possible, since overtraining seriously degrades continual learning performance. This can be more difficult than in supervised learning, because techniques like cross-validation are not always applicable (in the case of problems involving a single monolithic environment, for example).

In some circumstances however, catastrophic forgetting may be even more of an issue for neural net based RL agents than for supervised networks. As Baddeley discovered, and as this research subsequently confirmed, CF is a problem even for RL agents solving non-continual episodic problems, whereas CF is only a significant problem for supervised networks solving continual problems. As we have seen, avoiding this interference can speed up RL learning significantly.

scale to complex problems. Therefore, a more sophisticated approach is desirable (like a neural network, for example).

[19] In theory, any supervised problem can be rephrased as an RL problem, simply by changing (limiting) the feedback to the agent. In practice, doing so would be counterproductive and much slower than supervised learning, so the types of problem encountered in either domain are different by necessity.

Fortunately, there are several ways to reduce or entirely prevent catastrophic forgetting in RL problems, and in particular continual/lifelong RL problems. Before even attempting to use a continual learning algorithm, CF can be reduced by avoiding over-training, and by orthogonalising the input layer as much as reasonably possible and practical. The relatively simple Context Biasing algorithm can be used to modestly improve continual learning performance and training times.

For even better results, pseudorehearsal can be adapted into an RL agent, and it can completely eliminate CF. However, care must be taken in configuring the pseudorehearsal algorithm for use in RL, because small changes can have a large effect on performance. Finally, although the pseudorehearsal agent is a powerful continual learner, CHILD from Ring (1994) is still a faster and more capable continual learner. On the other hand, CHILD is disproportionately complicated and difficult to implement; so much so that the author was unable to find any other example of a working version of the algorithm detailed in published literature (despite several researchers noting its apparent sophistication).

5.2 Future work and extensions

Pseudorehearsal allows a standard neural network RL agent to be used successfully as a continual learning agent, and it may even speed up non-continual reinforcement learning. Despite its success, it still lacks some of the flexibility that a true lifelong learning intelligent agent needs. However, care should be taken to avoid unnecessarily complicating the agent, because over-optimisation can make an algorithm more difficult for other researchers to use, contrary to the aims of this project.

5.2.1 Scaling to more difficult problems

Although the problems implemented in this research are relatively simple with only a few dozen distinct states each, it still takes a significant amount of time for agents to learn to solve them by reinforcement (especially when the results need to be repeated to find a representative average). Even in these problems, the neural network agents are noticeably slower than a tabular or TTH agent, and they seem disproportionately slower when solving the harder problems from scratch. The main problem seems to be that the neural agents are especially bad at tackling the larger problems without any previous experience, because they do not seem disproportionately slower in the continual learning tests, when there is some prior knowledge to build on.

The trend is likely to continue for even harder problems, and this should also be investigated. Considering the extremely poor performance of some of the neural networks trying to learn the final maze in Section 4.8.1 from scratch, it will probably be necessary to bootstrap neural network agents with some sort of prior knowledge, even on non-continual and episodic problems. One innovative method for this is discussed in Rountree (2007), section 3.6.2. Unfortunately, decision trees are a batch learning algorithm and could be difficult to adapt to work in an RL agent, which typically requires an online system to learn utility values. However, for at least some problems it could be possible to use a tabular agent to quickly generate some prior knowledge. Just as a decision tree can initialise a supervised neural network, the neural network utility function of an RL agent could be initialised by teaching some or all of the mappings acquired by the tabular agent to a neural net utility function directly, by supervised learning.

5.2.2 "Lifelong" continual learning

The basic pseudorehearsal RL agent explored in Chapter 4 uses a fixed and pre-defined neural architecture. However, for lifelong learning it is unreasonable to expect to always be able to predict the maximum needed size of the network upon its creation. Additionally, it is inefficient to allocate more computing resources than are needed initially, and using an overly large network can lead to over-fitting to the dataset or environment.

A more general approach to lifelong learning is to add a constructive neural architecture to the RL pseudorehearsal agent. The Temporal Transition Hierarchies algorithm used by CHILD is constructive, so the idea works well in theory. The simplest way to accomplish this would be to incorporate the work of Eastman (2005), which extended the use of pseudorehearsal to a constructive neural algorithm (specifically, Cascade Correlation). However, Eastman found the algorithm tricky to develop and reported unstable results, so further research is needed.

5.2.3 Solving arbitrarily complex POMDPs

In addition to employing a practically unbounded constructive memory, intelligent agents need to be able to deal with ambiguous sensory data. In the field of RL, ambiguous hidden state problems are normally studied as POMDPs. To solve a POMDP, agents need to make use of some amount of recent or historical sensory inputs that they have received to disambiguate the current input. The RL pseudorehearsal agent is able to solve POMDPs by using delay lines to include recent sense inputs in the current sensory window, but it has several limitations. Using delay lines requires that we know the exact number of previous states we need to disambiguate, which is not realistic. However, adding to the context unnecessarily quickly increases the size and complexity of a network.

CHILD does not suffer from these limitations because the TTH algorithm incorporates a dynamic context window, created from the hierarchical structure of the algorithm. The RL pseudorehearsal agent could incorporate a dynamically increasing context window by incrementing its size whenever the current context was determined to be insufficient. However, using a dynamically increasing context is nearly as tricky as using a fully constructive architecture and it is hard to balance efficiently finding the right size with avoiding runaway increases due to elements during learning.

5.3 Summary

This research investigated in detail continual reinforcement learning problems, which (relatively speaking) have been less extensively studied than their supervised counterparts. When we implemented agents with neural network utility functions to solve problems of this type, we found that they suffered from catastrophic forgetting just as supervised neural networks do. In an attempt to overcome the problem, we implemented (in a Q-learning agent) and compared three types of neural network algorithms designed for continual learning: Temporal Transition Hierarchies (TTH), context biasing, and pseudorehearsal.

Our results show that TTH agents are the most efficient at this kind of task, but they are also by far more complicated and difficult to implement than pseudorehearsal, which is still efficient enough to be useful for practical purposes. Even though we found it has some unique complications when used in the context of an RL agent, pseudorehearsal is by contrast a straightforward modification to a well-understood and widely recognised algorithm. For these reasons, we believe that pseudorehearsal is more likely to be useful for reinforcement learning and intelligent agent research. To help foster potential collaboration, the agents and environments in Chapter 4 were all developed with RL-Glue, an open source framework developed to improve collaboration among RL community researchers.

Bibliography

Ans. (2004). Sequential Learning in Distributed Neural Networks without Catastrophic Forgetting: A Single and Realistic Self-Refreshing Memory Can Do It. *Neural Information Processing - Letters and Reviews 4(2)* .

Ans, B., Rousset, S., French, R. M., & Musca, S. (2002). Preventing Catastrophic Interference in Multiple-Sequence Learning Using Coupled Reverberating Elman Networks. *Proceedings of the 24th Annual Conference of the Cognitive Science Society* (pp. 71-76). Hillsdale NJ: Lawrence Erlbaumn.

Ans, Rousset, S., Musca, S., & French, R. (2003). Self-refreshing memory in artificial neural networks: Learning temporal sequences without catastrophic forgetting. *Cognitive Science* .

Baddeley, B. (2008). Reinforcement learning in continuous time and space: Interference and not ill conditioning is the main problem when using distributed function approximators. *IEEE transactions on systems, man, and cybernetics - Part B, Vol 38, No 4* , 950-956.

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

Boyan, J., & Moore, A. (1995). Generalization in reinforcement learning: Safely approximating the value function. *Advances in Neural Information Processing Systems 7* .

Chapman, D., & Kaelbling, L. P. (1991). Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. *Proceedings of the International Joint Conference on Artificial Intelligence.* Sydney, Australia.

Deco, G., & Obradovic, D. (1996). *An Information-Theoretic Approach to Neural Computing.* New York: Springer-Verlag.

Eastman, T. V. (2005). *Using constructive neural networks for serial learning.* Dunedin: University of Otago.

Frean, M., & Robins, A. (1999). Catastrophic forgetting in simple networks: an analysis of the pseudorehearsal solution. *Computation in Neural Systems, Volume 10, Issue 3* , 227-236.

Frean, M., & Robins, A. (1998). Learning and generalisation in a stable network. *Progress in Connectionist-based Information Systems: Proceedings of the 1997 Conference on Neural*

Information Processing and Intelligent Information Systems (pp. 314-317). Singapore: Springer-Verlag.

French. (1999). Catastrophic Forgetting in Connectionist Networks: Causes, Consequences and Solutions. *Trends in Cognitive Sciences 3(4)* , 128-135.

French. (1994). Dynamically constraining connectionist networks to produce distributed, orthogonal representations to reduce catastrophic interference. *Proceedings of the 16th Annual Cognitive Science Society Conference* (pp. 335-340). NJ: LEA.

French. (1991). Using Semi-Distributed Representations to Overcome Catastrophic Forgetting in Connectionist Networks. *Proceedings of the 13th Annual Cognitive Science Society Conference* , 173-178.

Gurney, K. (2007). Neural networks for perceptual processing: from simulation tools to theories. *Philosophical transactions of the royal society B* , 339-353.

Hamker, F. H. (2001). Life-long learning Cell Structures-continuously learning without catastrophic interference. *Neural Networks 14* , 551-573.

Hongxing, L., Jiayin, W., Yundong, G., & Yanbin, F. (2004). Hardware implementation of the quadruple inverted pendulum with single motor. *Progress in Natural Science, Volume 14, Issue 9* , 822-827.

Kaebling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research 4* , 237-285.

Kanerva, P. (1988). *Sparse Distributed Memory.* MA: MIT Press.

Kruschke, J. K. (1991, February 22). ALCOVE: An exemplar-based connectionist model of category learning. *Indiana University Cognitive Science Research Report 19* .

Lee, E., & Perkins, J. (2008). *Comparison of techniques for stabilization of a triple inverted pendulum.*

Lin, L. J. (1991). Programming robots using reinforcement learning and teaching. *Proceedings of the Ninth National Conference on Artificial Intelligence.*

Lin, L. J. (1992). Self-Improving Reactive Agents Based On Reinforcement Learning, Planning and Teaching. *Machine Learning, 8* , 293-321.

McCallum, S. (2007). *Catastrophic Forgetting and the Pseudorehearsal Solution in Hopfield Networks.* Dunedin: University of Otago.

Mitchell, M. W. (2003). *Using Markov-k memory for problems with hidden state.* Victoria, Australia: Monash University.

Provost, J. (2007). *Reinforcement Learning in High Diameter, Continuous Environments.* Austin: University of Texas.

Ring, M. (1994). *Continual Learning in Reinforcement Environments.* Austin: University of Texas.

Rivest, F., & Precup, D. (2003). Combining TD-learning with Cascade-correlation Networks. *Proceedings of the Twentieth International Conference on Machine Learning* (pp. 632-639). Washington DC: AAAI Press.

Robins. (1995). Catastrophic forgetting, rehearsal, and pseudorehearsal. *Connection Science: Journal of Neural Computing, Artificial Intelligence and Cognitive Research* , 123-146.

Robins. (2004). Sequential learning in neural networks: A review and a discussion of pseudorehearsal based methods. *Intelligent Data Analysis 8(3)* , 301-322.

Robins, A., & McCallum, S. (1998). Catastrophic forgetting and the pseudorehearsal solution in Hopfield type networks. *Connection Science: Journal of Neural Computing, Artificial Intelligence and Cognitive Research, 10* , 121 - 135.

Rountree, N. (2007). *Initialising Neural Networks with Prior Knowledge.* Dunedin: PhD thesis, University of Otago.

Rumelhart, D. E., & McClelland, J. L. (1986). *Parallel distributed processing, exploration in the microstructure of cognition - Vol. 1: Foundations.* Cambridge: MIT Press.

Russell, S. J., & Norvig, P. (2003). *Artificial Intelligence: A Modern Approach* (Second Edition ed.). New Jersey: Pearson Education.

Schmidt, R. (2005). *Exploration and extension of temporal-difference models of midbrain dopamine cell firing.* Dunedin: University of Otago.

Shang, Y., & Wah, B. (1996). Global optimization for neural network training. *IEEE Comput. 29* , 45-54.

Sharkey, N., & Sharkey, A. (1995). An analysis of catastrophic interference. *Connection Science, 7* , 301-329.

Sutton, R. S., & Barto, A. G. (2005). *Reinforcement Learning: An Introduction.* Cambridge: The MIT Press.

Tanner, B. (2008, December 1). *RL Community*. Retrieved December 1, 2008, from RL Community: http://www.rl-community.org/

Tesauro, G. J. (1994). TD-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation 6(2)* , 215-219.

Thrun, S. (1994). A Lifelong Learning Perspective for Mobile Robot Control. *Proceedings of the IEEE/RSJ/GI Conference on Intelligent Robots and Systems.*

Thrun, S., & Mitchell, T. (1995). Learning One More Thing. *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence.* San Mateo, CA: Morgan Kaufmann.

Wilson, D. R., & Martinez, T. R. (2003). The general inefficiency of batch training for gradient descent learning. *Neural Networks, vol. 16, no. 10* , 1429-1451.

Appendix A: French94 Dataset

Original Records																	Maverick Records																
D	0	1	1	0	1	1	0	0	0	0	0	0	1	1	1	1	**D**	0	1	0	1	1	1	0	0	0	0	0	0	1	1	1	1
D	1	1	1	0	0	0	1	1	1	0	1	0	0	0	1	1	**D**	0	1	0	1	0	0	0	0	0	0	1	0	0	0	1	1
D	1	1	1	0	0	0	1	1	1	0	0	0	0	0	1	1	**D**	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	1
D	1	0	1	0	0	0	1	1	1	1	0	0	0	0	1	1	**D**	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	1
D	1	0	1	0	0	0	1	1	1	0	1	0	0	0	1	1	**D**	0	0	0	1	0	0	0	0	0	0	1	0	0	0	1	1
D	1	1	1	0	0	0	1	1	1	0	1	0	0	0	1	1	**R**	1	0	1	0	1	0	1	1	1	0	0	1	1	1	0	1
D	1	1	1	0	0	0	1	1	1	0	1	0	0	0	1	1	**R**	1	1	1	0	1	1	1	1	1	0	0	1	1	1	0	0
D	1	1	1	0	0	0	1	1	1	1	0	0	1	0	1	1	**R**	1	1	1	0	1	1	1	1	1	0	0	1	1	1	0	1
D	1	1	1	0	0	0	1	1	1	0	0	0	0	0	1	1	**R**	1	1	1	0	1	1	1	1	1	0	0	1	1	1	0	0
D	1	0	1	0	0	0	1	1	1	1	1	0	1	0	1	1	**R**	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	1
D	1	0	1	0	0	0	1	1	1	0	0	0	0	0	0	1																	
D	1	0	1	0	0	0	1	1	1	0	0	0	0	0	1	1																	
D	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	1																	
D	1	1	1	0	0	0	1	1	0	0	0	0	0	1	0	1																	
D	1	1	1	0	0	0	1	1	1	0	1	0	0	0	1	1																	
D	1	1	1	0	0	0	1	1	1	0	1	0	0	0	0	1																	
D	1	1	1	0	0	0	1	1	1	0	0	0	0	0	1	1																	
D	1	0	1	0	0	1	1	1	1	1	1	0	0	1	1	1																	
D	0	1	1	1	1	1	0	0	0	1	1	0	1	1	0	0																	
D	0	1	1	1	1	1	0	1	1	1	1	1	1	1	0	1																	
D	1	1	1	0	1	1	0	0	0	1	1	0	1	1	0	1																	
D	0	0	1	0	1	1	0	0	0	1	1	1	1	1	0	1																	
D	0	1	1	0	1	1	1	0	1	1	1	0	1	1	0	1																	
D	1	0	1	0	0	0	1	1	1	1	0	0	0	0	1	1																	
D	1	0	1	0	0	0	1	1	1	1	1	0	0	0	1	1																	
R	0	1	0	1	1	1	0	0	0	0	0	1	1	1	0	1																	
R	1	0	0	1	1	0	1	1	1	0	0	1	1	1	0	1																	
R	0	1	0	1	1	1	0	0	0	0	0	1	1	1	0	0																	
R	1	1	0	1	1	1	0	0	0	0	0	0	1	1	0	1																	
R	1	1	0	1	1	1	0	0	0	1	0	1	1	1	0	0																	
R	0	1	0	1	1	1	0	0	0	1	1	1	1	1	0	0																	
R	0	1	0	1	1	1	0	0	0	0	0	1	1	1	0	0																	
R	0	1	0	1	1	1	0	0	0	0	0	1	1	1	0	1																	
R	0	1	0	1	1	1	0	0	0	1	0	1	1	1	0	0																	
R	0	1	0	1	1	1	0	0	0	1	1	1	1	1	0	1																	
R	0	1	0	1	1	1	0	0	0	1	0	1	1	1	0	1																	
R	0	1	0	1	1	1	0	0	0	1	0	1	1	1	0	1																	
R	1	1	0	1	1	1	1	0	0	0	0	1	1	1	0	1																	
R	0	1	0	1	1	1	1	0	0	0	1	1	1	1	0	1																	
R	0	1	0	1	1	1	0	0	0	0	0	1	1	1	0	0																	
R	0	1	0	1	1	1	0	0	0	1	0	1	1	1	0	0																	
R	1	1	1	1	0	0	1	1	1	1	1	0	0	1	0	1																	
R	1	0	1	1	1	0	1	0	1	1	0	0	1	1	0	1																	
R	0	0	0	1	1	0	0	0	0	1	0	1	1	1	0	0																	
R	0	0	0	1	1	1	0	0	0	1	0	1	1	1	0	1																	
R	0	0	0	1	1	1	0	0	0	0	0	1	1	1	0	0																	
R	0	0	0	1	1	1	0	0	0	1	0	1	1	1	0	0																	
R	0	0	0	1	1	1	0	0	0	0	0	1	1	1	0	0																	
R	0	1	0	1	1	1	0	0	0	0	0	1	1	1	0	1																	
R	1	1	1	1	1	0	1	0	0	0	0	1	1	1	0	1																	