REINFORCEMENT LEARNING USING ASSOCIATIVE MEMORY NETWORKS

by

Ricardo Salmon

Bachelor of Science, Computer Science, Ryerson University, 2007

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Science

in the Program of

Computer Science

Toronto, Ontario, Canada, 2009

I hereby declare that I am the sole author of this thesis or dissertation.

I authorize Ryerson University to lend this thesis or dissertation to other institutions or individuals for the purpose of scholarly research.

Signature: _____ Date: _____

I further authorize Ryerson University to reproduce this thesis or dissertation by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Signature: _____ Date: _____

REINFORCEMENT LEARNING USING ASSOCIATIVE MEMORY NETWORKS

Ricardo Salmon

Master of Science, Computer Science, Ryerson University, 2009

It is shown that associative memory networks are capable of solving immediate and general reinforcement learning (RL) problems by combining techniques from associative neural networks and reinforcement learning and in particular Q-learning. The modified model is shown to outperform native RL techniques on a stochastic grid world task by developing correct policies. In addition, we formulated a analogous method to add feature extraction as dimensional reduction and eligibility traces as another mechanism to help solve the credit assignment problem. The network contrary to pure RL methods is based on associative memory principles such as distribution of information, pattern completion, Hebbian learning, and noise tolerance (limit cycles, one to many associations, chaos, etc). Because of this, it can be argued that the model possesses more cognitive explanative power than other RL or hybrid models. It may be an effective tool for bridging the gap between biological memory models and computational memory models.

Acknowledgements

I would like to thank my supervisor Alireza Sadeghian for his supervision and allowing me the independence to research a broad range of ideas. Furthermore, without him this thesis would not be possible.

In addition, I would like to thank my other advisor Sylvain Chartier from the University of Ottawa for his help from his knowledge, background and suggestions.

Also, I would like to thank Eric Harley for his help in getting me involved in my research area from his course and recommendations.

Finally I would like to thank the department and my supervisor for all financial support provided.

Table of Contents

List of Abbreviations

AM	Associative Memory
BAM	Bidirectional Associative Memory
RL	Reinforcement Learning
FEBAM	Feature Extracting Bidirectional Associative Memory
kWTA	k Winners Take All
MDP	Markov Decision Process
DP	Dynamic Programming
MC	Monte Carlo
TD	Temporal Difference
Q(λ)	Q-learning with Eligibility Trace
VPTF	Value function, Policy, eligibility Trace, Feature extraction

List of Tables

List of Figures

Chapter 1

Introduction

Associative memory (AM) can be seen as a possible computational model of the brain and human control for the reason that it appears to be one of the most important functions in many cognitive processes. Many brain structures can be modeled as associative memories. This is evident by our remarkable ability at pattern recognition and pattern completion which AM networks excels at. We know the brain is capable of many forms of learning and specifically supervised and unsupervised learning. If these theories are correct, then our models must also be able to perform the various learning tasks a human could do including reinforcement learning.

Supervised learning can be thought of as being taught the correct action from a teacher. However, for unsupervised learning, no correct action is given, instead, the agent must establish order on its own by some measurement of desirability. In contrast, reinforcement learning sits between both in that there is no teacher available with the correct action, but a response is received from the environment that quantify the desirability of taking that action in some state. A case can be made that associative memories are well suited to model human learning because of their dynamical properties. These include the ability to exhibit attractor behaviour such as fixed-points, limit cycles and strange attractors which are essential to dealing with noisy inputs and have been supported by Sharda and Freeman [3, 4] as fundamental to the way the brain stores and recalls information.

1.1 Problem

The problem this work attempts to address is to allow AM networks to perform reinforcement learning on pattern sequences with associated rewards. Pattern sequences can be seen as states changing over time in response to some actions taken in the environment. Therefore, anticipating and maximizing the rewards over states by taking the right action are in general the reinforcement learning (RL) problem. For this to be feasible, we will build on previous work in both the associative memory and reinforcement learning problem that includes solving the temporal credit assignment problem. The temporal credit assignment problem is to determine which actions are responsible for certain behaviours from a sequence of actions by using states, actions, rewards and optionally a model of the environment. There are quite a few proposed solutions and techniques within the reinforcement literature that we will leverage to our advantage such as the Q-Learning algorithm [5] and Eligibility traces [6, 7, 8] together called $Q(\lambda)$.

1.2 Motivation

So far the AM networks has been applied to performing supervised and unsupervised learning task. The importance of this work is that it will provide a unified model of learning that is capable of three types of learning: supervised, unsupervised, and reinforcement learning. This is sought after because cognitive science researchers are looking for solutions to apply to multiple perceptual and cognitive tasks without assuming an arbitrary number of rules and architectures. The significant contribution would be a working model of the associative memory architecture applying reinforcement learning solutions to solve temporal decision problems.

The reason we are tackling this problem is because we find the biological ramifications exciting and the dynamical nature of the model interesting enough to warrant the effort. Furthermore, the applications in psychology and neuroscience from having a unifying model of learning based on associative memory networks. The explanatory power of the model would be valuable on it's own in addition to robotics control, control problems, and planning systems. These would be ideal applications especially when the problem

domain is new in cases such as large scale problems where a good enough solution is desired over a computationally optimal method that might not be feasible or when tabular representation would be too costly storage wise or the problem cannot be represented by a small set of key features in this case we could attempt to reduce its dimension. The importance of this work, however, is to investigate the feasibility of using Hebbian [9] inspired learning rule to solve reinforcement problems. Our objective, therefore, is to derive a reinforcement learning system by using associative memory network principles while maintaining its advantages.

By using the network state, we were able to store arbitrary values in the network as the energy of that particular state, and use these values to make further estimates. These were used as a basis to simulate Q-learning action values within the AM network and successfully solved a stochastic Gridworld problem and a modified game of Tetris. Q-learning is a common algorithm used solve reinforcement learning problems. Moreover, we already know that associative networks can store sequences using limit cycles encodings [10] as well as one to many associations [2]. Furthermore, recent work by [11] has shown progress in using chaos as a search process through state space. This works by adjusting a parameter of the network which has the effect of influencing the output of the network from completely chaotic to chaos bounded within a region to fixed-point attractors. It is proposed that this value could be reduce as we enter more desirable regions until the network converges to a fixed-point. We can simulate a searching process using the dynamics of the network itself instead of incorporating a stochastic element into the network. The purpose behind this is to simulate exploration in the network which is required to learn optimal policies from experiences (i.e., ϵ-greedy polices). In addition this seems like a more biologically plausible technique than more complex searching algorithms from the perspective of associative networks.

1.3 Related Work

In the past few years researchers have been exploring the idea of inserting prior knowledge into bidirectional associative memory (BAM) neural networks. One method by Chartier

[11] involved using an asymmetry parameter in the output function of the network. This has the desired effect of biasing the state space of the network into particular region for a moment during recall. Furthermore, this asymmetry parameter can be set and controlled by the environment or controlled by a second network trained by a supervised learner.

For instance, the XOR problem has been solved using this method by Chartier by a collection of BAMs, which was never achieved before using Hebbian learning. We first associate each input and output pairs as usual. Then when the model failed to associate the right pair the environment would suggest a different value for the asymmetry parameter until it has perform the correct association. Then the parameter value would be used for future recalls. The process has many similarities with reinforcement learning, however, only immediate success or failure is possible to learn at the moment.

Another project by Zhu [12] was based off Palm networks. These are a special kind of associative memory model with neuron connections and neurons output only taking binary values. Zhu used two networks, one to represent the policy, the other to represent the value function. By iteratively updating both networks as a function of the other it was able to converge to the correct policy. This system was shown to perform well on a deterministic gridworld task against Q-learning.

1.4 Limitations and Key Results

Our proposed model was compared with two RL methods on a stochastic gridworld problem where the task was to find the optimal path to a correct terminal state. In this task, our model outperformed both methods by a significant margin and validated our objective that AM can be used to solve RL problems. A limitation of our method is that we were not able to successfully use the dynamic policy as an optimal action selection procedure for both greedy and exploratory actions. Then on a modified game of Tetris the model's performance over various ranges of the parameters was analyzed including a pure greedy policy where we see significant improvements within two episodes. In particular, we use the ability to extract features from the original problem space to a lower dimension and use this as a basis to make decisions. The purpose of doing this was

the reduced number of operations required and in effect trading time for accuracy.

1.5 Outline

In the next chapter we will discuss the necessary background knowledge for our work and any related work from the literature. In particular we will cover associative memory, chaos in associative networks with applications to neuroscience, the theory of reinforcement learning and related work using associative memories. Afterwards we will present our work in chapter 3 with our contributions to the field and what we have accomplished. In chapter 4 we will explore and analyze the effectiveness of our work on two problems from the literature. The results will be analyzed and compared against prior work in the reinforcement learning literature. Finally to conclude with we will summarize our contributions and discuss possible issues and areas of future work that were not covered in this document.

Chapter 2

Literature Review

Associative memories are an important class of neural network models that have seen strong growth since its original inception in a wide scope of applications. These models are a convenient tool for modeling human thinking and intelligent systems. Associative memories can be divided into two categories: Autoassociative and Heteroassociative. In the following sections, we will briefly go into detail regarding chaos, neuroscience and reinforcement learning.

2.1 Autoassociative Memory

Since Hopfield's original paper[13] there has been significant interest in computational models of the brain for applications in control systems, optimizations, knowledge representation and neuroscience. The autoassociative networks are one of the first well-understood recurrent neural networks.

These networks are constructed from a large numbers of identical neurons (also called units) based on the Ising model from physics. In these networks, the input and output units are identical. However, collectively the model is able to recover data given a partial sample of sufficient size from the input space. In addition emergent properties such as ability to generalize, familiarity recognition, pattern completion, categorization, noise reduction and error correction were possible. This work was later extended [14] to the continuous real valued states.

The autoassociative network is described by a collection of n binary or bipolar state units that are connected to every other unit except itself and weights are symmetric, i.e., $w_{ii} = 0$ and $w_{ij} = w_{ji}$. Figure 2.1 is an illustration of a 4-state autoassociative network.

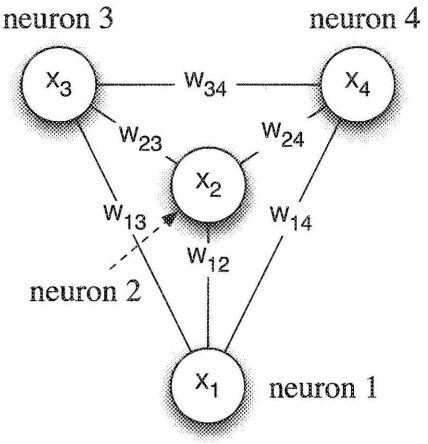

Figure 2.1: Four state Hopfield network.

The constraints on the structure of the network guarantee convergence to a local minimum, however, this does not rule out undesired patterns. The lack of self-connections in each unit is necessary to avoid the potential periodic or chaotic behaviour in the transition between states while symmetric connections are necessary to guarantee convergence. The output of each unit is deterministically calculated by whether the sum of all its input exceeds a certain threshold θ as $x_{t+1} = sgn(x_t\mathbf{W})$ for

$$sgn(a_i) = \begin{cases} +1 & \text{if } a_i > \theta_i \\ 0 & \text{if } a_i = \theta_i \\ -1 & \text{if } a_i < \theta_i \end{cases} \tag{2.1}$$

where $\mathbf{W}$ is the weight matrix, x is the unit state, $a_i = x_i\mathbf{W}$ the activation of unit i, and θ_i is the threshold of neuron i. The output units have been shown to converge in asynchronous mode as well [15], i.e., when each unit's state is updated independently of each other. The computation is complete when the system has settled to a stable pattern such that $x = sgn(x\mathbf{W})$. The particular pattern the network settles to is referred to as the attractor.

Hopfield's description of computation within this model was to imagine a physical dynamical system and each memory corresponds to a stable state of the system. Isomorphic to an Ising model a scalar quantity E can be used to describe the state of the system known as the energy. The energy is proportional to the degrees of freedom available at each state:

$$E(\mathbf{X}) = -\mathbf{X}^T\mathbf{W}\mathbf{X} \tag{2.2}$$

It can be shown that the energy of the system decreases or remains the same for each state update and fixed points corresponds to local minima of the energy function [13]. Furthermore, the recalling procedure would be to present a pattern x to the network and a search through state space would coincide with following the gradient of the energy function to the lowest energy state x^* from the initial pattern. Illustrated in Figure 2.2 is the energy function of a 2-dimension hopfield network with states {[1,1],[1,-1],[-1,1],[-1,-1]}. If we were to start in one of the higher states of [1,-1] or [-1,1] we would decent to a lower state of either [1,1] or [-1,-1]. In the search process, a random neuron can be updated at each time step. This stable state x^* would represent the output of the system.

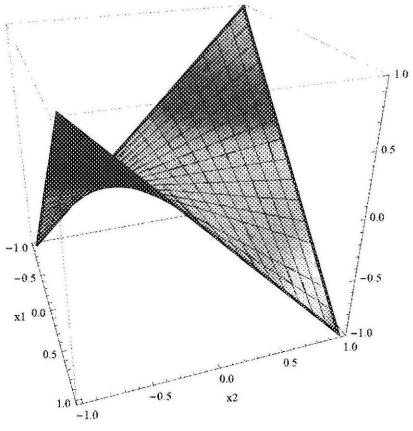

Figure 2.2: Energy function of a two state flip-flop with weight set to -1.

Training can be achieved using any method capable of making arbitrary points in the system stable states by lowering the energy level of those states. For example, a memorization of vector $(1, -1, -1, 1)$ would have the effect of pulling on near by vectors such

as $(1,-1,1,1)$ and $(1,1,-1,1)$. The network can also be trained using the Perceptron learning [16] or genetic algorithms [17]. However, a common approach is to use Hebbian learning [9] and update the weights (initially set to zero) according to the rule:

$$w_{ij} \leftarrow w_{ij} + x_i^p x_j^p, i, j = 1, \ldots, n \text{ and } i \neq j \tag{2.3}$$

for each pattern p to be stored in the network but this can also be written in matrix form as

$$\mathbf{W} = x_1^T x_1 + x_2^T x_2 + \ldots + x_m^T x_m - m\mathbf{I} \tag{2.4}$$

where m is the number of patterns. We do a subtraction at the end to remove the sum in the diagonals that corresponds to self connections. However, the storage capacity of an n dimensional space is limited to maximum of n memories or fix points. It is known from the matrix $\mathbf{W}$ eigenvectors that to achieve this maximum requires n orthogonal n-dimensional vectors. Without orthogonal vectors we have undesirable attractors that are formed from interaction with other patterns and some quite arbitrarily. Furthermore, from the system of equations we have

$$sgn(-x\mathbf{W}) = -sgn(x\mathbf{W}) = -x \tag{2.5}$$

indicating that using Hebbian learning with a symmetric weight matrix will always produce the pattern and its complement as attractors in the network in addition to linear and non-linear combinations of our initial patterns. The limitation of orthogonal vectors can be relaxed by using the pseudo-inverse algorithm for training instead. Unfortunately the pseudo-inverse is non-iterative and non-local, that is, each unit can no longer be arbitrarily updated based only on the sum of its neighbours.

In addition to the abilities mention, the autoassociator networks are able to perform optimization computations. To do so, the optimization task can be written in a form isomorphic to the energy function. So the difficulty is in encoding the constraints of the problem in the connections of the network and traversing the state space to the least energy. Local minimum in the energy function will correspond to sub-optimal solutions

and the global minimum to optimal solutions. Such encoding has been shown to find near optimal solutions on the traveling salesman problem[18] and with better results by [15] on the multi-flop problem, eight rooks problem, and eight queens problem.

However, this method does not provide a way of solving NP problems in polynomial time since the number of units required to solve each task scales exponentially compare to the size of the problem [15]. In addition, there is no guarantee of finding the optimal solution while using gradient decent to drive the searching process as the number of local solutions grows so large. This motivated research into other methods, in particular Boltzmann machines. Boltzmann machines use the concept of a temperature from statistical mechanics to allow movement to higher energy state proportional to the temperature [19]. As the temperature slowly cools over time the probability of jumping to higher energy states decreases.

2.2 Heteroassociative Memory

Later, Kosko [10] proposed a bidirectional associative memory (BAM) model that associates pair of patterns. BAM is the link between unsupervised and supervised models. The network architecture can be viewed as a bipartite graph and the two groups are the pair (x, y) that feedback to the other layer. Probing the network with x will retrieve the pattern y and visa versa for y will retrieve x

$$y = sgn(x\mathbf{W}) \text{ and } x = sgn(\mathbf{W}^T y) \tag{2.6}$$

The units used are identical to the autoassocative model with either binary or bipolar states and symmetric connections without any self-connection. A simple Hebbian learning rule is employed as before but with vector pairs x and y

$$\mathbf{W} = x_1^T y_1 + x_2^T y_2 + \ldots + x_m^T y_m \tag{2.7}$$

The same reasoning for the autoassociator can be used to show that the energy function of the network can be describe by the function

$$E(\mathbf{X}, \mathbf{Y}) = -\mathbf{X}^T\mathbf{W}\mathbf{Y} \tag{2.8}$$

If $x = y$ the BAM collapses into a Hopfield network of equal dimensions and similar capabilities. The BAM is also able to store complex spatial-temporal patterns that are equivalent to Grossberg outstar avalanche coding [20] such as limit cycles. The temporal coding works on a sequence of patterns

$$(x_1, x_2, x_3, \ldots, x_m)$$

by training the pairs

$$(x_1, x_2), (x_2, x_3), \ldots, (x_{m-1}, x_m), (x_m, x_1)$$

on the network.

The issue of binary/bipolar encoding has been dealt with in [10] where it was shown that bipolar coding is better than binary coding in terms of strength and sign of the correction term coefficients.

Since Kosko's paper there has been many variants that tries to overcome its initial weaknesses in storage capacity [21, 22], expressive power, biological plausibility, symmetric relations and recall performance. These approaches varied in sophistication from modifying the architecture, learning rule, and output function.

Changes to the architecture include adding a second layer with its own separate weight matrix [2] that recurrently sends information to each other in synchronization. This implies that the weight matrices need not to be transposes of each other. This model tries to find solutions to the non-linear equation of the form:

$$\mathbf{X} = f(\mathbf{V}\mathbf{Y}) \text{ and } \mathbf{Y} = f(\mathbf{W}\mathbf{X}) \tag{2.9}$$

for an output function f. By using a separate weight matrix, the network is now able to perform task such as feature extraction [23], categorization [24] and other common unsupervised task [25].

If we add a context unit to the BAM for sequence recognition we will be able to perform one to many associations. Each additional context unit allows us to do more associations with longer context. While learning the XOR problem, a second BAM can be used to provide additional information to the primary BAM by associating the desired bias for the correct pattern [26]. This will be relevant later for immediate reinforcement learning.

An improved learning rule by Haykin [27], i.e., Hebbian cross correlation learning based on Hebbian (but with an anti-Hebbian correction term) can be expressed:

$$\mathbf{W}_{k+1} = \mathbf{W}_k + \eta(y_0 - y_t)(x_0 - x_t)^T \tag{2.10}$$

$$\mathbf{V}_{k+1} = \mathbf{V}_k + \eta(x_0 - x_t)(y_0 - y_t)^T \tag{2.11}$$

where $\mathbf{W}$ and $\mathbf{V}$ represent the weight matrix, x_0 and y_0 are the original patterns to be learned, x_t and y_t are the patterns iterated through the network t times and η is the learning parameter. An autoassociative learning rule can also be derived from these equation if we let $x_0 = y_0$ this reduces to a much simpler rule:

$$\mathbf{W}_{k+1} = \mathbf{W}_k + \eta(x_0 x_0^T - x_t x_t^T) \tag{2.12}$$

In fact by removing the initial connections for y_0 we can get a new architecture [23, 24] with feature extracting (FEBAM) abilities similar to principal component analysis in addition if the network output is limited to bipolar states then clustering or categorization is performed. The process to derive the vectors necessary for training are illustrated in Figure 2.3. To start training, the weights are initially set to small random values and the output of the initial vector x_0 is used as the initial vector y_0. In this network, the smaller the dimension of y relative to x the greater compression is achieved for the extracted features.

We can go even further by constraining the amount of units that are able to fire in the compressed (y) layer. This would allow the network to simulate *k-winner-take-all* (kWTA) [28] characteristics and hence categorical behaviour and as k is increased

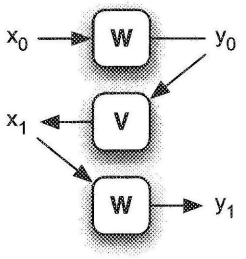

Figure 2.3: Update process without initial connections from y_0.

towards the number of units in y we get back our original network. Chartier and Giguere [25] were able to show the network achieve multiple categorizations and in addition being able to store exemplars. Also by constraining the y layer, the network was able to group similar topological patterns in a neighbourhood while also reducing the dimensions like *self-organizing feature maps.*

The advantage of these method over other rules such as Hebbian, optimal projection and the pseudo-inverse are online learning by incorporating feedback into the weight updates, maintaining the simplicity of the Hebbian rule and preserving locality. The algorithm is very simple in nature by reducing the error between the patterns and its state after being iterated through the network t times. A fixed-point attractor is reached iff the two terms are equal, i.e., $x_0 = x_t$ and $y_0 = y_t$.

A new non-linear output function [29] departing from the step function is:

$$x_{i+1} = f(a_i) = \begin{cases} +1 & \text{if } a_i > +1 \\ -1 & \text{if } a_i < -1 \\ (\delta + 1)a_i - \delta a_i^3 & \text{if } -1 \leq a_i \leq +1 \end{cases} \tag{2.13}$$

where a_i is the activation of unit i written $a_i = \mathbf{W}x_i$, x_{i+1} as the new state and δ as the transmission parameter. Similarly if we replace x_{i+1} by y_{i+1} and a_i by $\mathbf{V}y_i$. The parameters value should be set according to:

$$\eta < \frac{1}{2(1-2\delta)max(N,M)}, \delta \neq \frac{1}{2} \tag{2.14}$$

for guaranteed convergence where N and M are the number of units in each layer. The third condition in equation (2.13) allows the network to develop non-bipolar attractors for when $(\delta+1)a_i+R = \delta a_i^3$ for some real-valued constant R in addition to bipolar attractors. This is in contrast to previous models that were only capable of forming attractors on the corner of a hypercube in n-dimensions [30] or by using special encoding schemes. The result from these modifications is the BAM can now associate grey level patterns such as the various shades of an image [31]. Furthermore by changing the learning rule Xu el la [32] were able to infer asymmetric relations with a BAM.

In a typical autoassociative and heteroassociative system the network is trained with the following procedure:

1. weights are initially set to zero.
2. randomly select a pair of patterns.
3. iteration pattern through the network k times (common for $k = 1$) using equation (2.13).
4. update weights by using equations (2.10) and (2.11).
5. repeat steps 2-4 until desired error or number of trials is reached.

2.3 Chaos and Neuroscience

Although previous changes to the BAM were driven by performance, the inclusion of chaos was partly inspired by work from neuroscience [33]. There were mounting evidence that the brain is a dynamical system with inherit chaos. Stored patterns may be stationary, time-varying, quasi-chaotic or chaotic. Different attractors are associated with different functions such as memory, motor behaviour, and classification [34, 35].

Work by Freeman, Skarda and Kay [4, 3] uncovered that chaos is fundamental to odour perception in animals. The results from these findings suggest that animals do not directly respond to external stimuli but internal stimuli created from chaos in the olfactory bulb, olfactory cortex, and the hippocampus. This suggests a complex interplay

at both macroscopic and microscopic scales that is in direct contradiction with the widely held static view of memories, that is, memories exist merely as fixed-point attractors in the brain.

Kaneko and Tsuda [1] later described a so-called Chaotic Itinerancy (CI) as an explanation for how the brain stores memories. CI is essentially the result of destabilizing a system with a stable attractor such that the attractor is neither totally ordered or disordered. A consequence of these phenomena in high dimensional space is that strange unpredictable orbits form that give rise to states that switch between ordered motion and total chaos. Moreover, CI has been identified in many real world dynamical systems, e.g., coupled maps, optical turbulence, dynamics of water molecules, climate changes, animal brain, biochemical reaction dynamics in cells are among some of the areas studied.

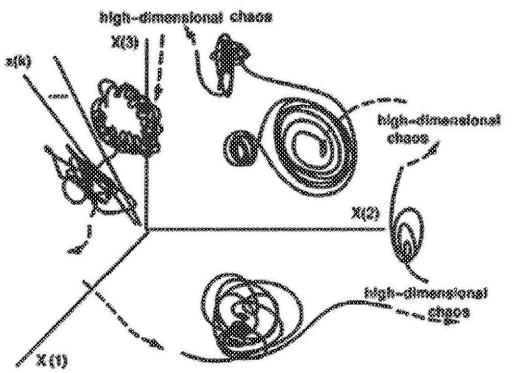

Figure 2.4: Diagram of itinerancy in high dimension space [1].

Situations equivalent to the ones described above could arise between attractor ruins (see Fig. 2.4). For some stable strange attractor in high dimensional space a small perturbation in the bifurcation parameter could cause instability of this attractor that causes it to fragment into many smaller attractors with chaotic orbits. A bifurcation diagram shows the fixed-points and periodic orbits of a system as a function of the bifurcation parameter in this case δ. Such groups of attractor are called attractor ruins. While in an attractor ruin the orbit is chaotic but eventually escapes to another attractor ruin. But in between, motion in high dimensional space is in some circumstances constraint to a low dimensional manifold. An escape is possible if not all orbits in the trajectory of the

basin of the attractor are attracting. The path taken is seen as a type of dynamic memory or a searching process. However, this path can be differentiated from noise since its past and future orbit is entirely deterministic. That is the distinguish chaotic behaviour from randomness we know a chaotic system always evolve the same way starting from any initial state. A basic method is to pick a starting state and find a near by state in the system and study how they evolve over time. In a stochastic system the difference in the two states will be randomly distributed while in a deterministic system the difference will be regular or increase exponentially for a chaotic system.

That's the idea behind Lyapunov exponents which are usually used to test for stability in a system by measurement of the separation between two adjacent states [36]. The Lyapunov exponent is negative for stable systems and positive for chaotic systems as shown in Figure 2.5.

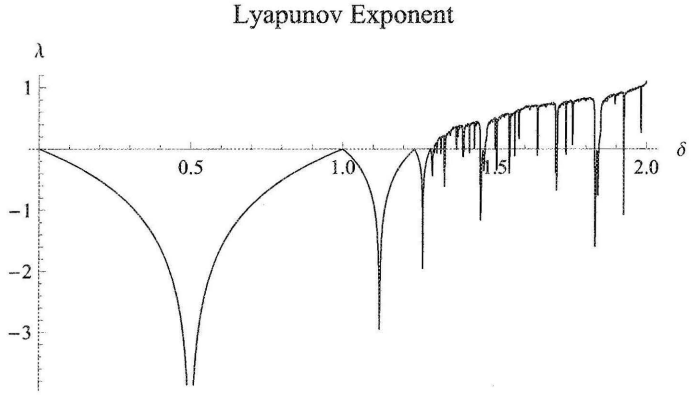

Figure 2.5: Lyapunov exponent for a 1D network with weight set at 1 and initial system state at 0.9 for 5,000 iterations.

Earlier effort of including chaos into the BAM are dominated by more complex neuron models. These models seek to replace the simple neuron model with a more complex model [37, 38] that is capable of many behaviours such as periodic response and deterministic chaos usually as a collection of differential equations or difference equations. Crisis behaviour is the result of adjusting some parameter of the network the system swaps behaviours from periodic to chaotic and vise versa. Auaujo et al [38] introduced a family of chaotic bidirectional associative memories (C-BAM) represented by three equations to describe the states of each neuron. This model uses four additional parameters that in

a certain range of values enable the network is able to show chaotic or crisis behaviour. However, when all the parameters are set to 0 the network naturally defaults back to a standard BAM [10]. This family includes modifications to Chaotic BAM, Delay BAM and Exponential C-BAM. Adachi and Aihara [37] were able to show that transient state space depends on initial conditions and that such states are characteristic of a searching process.

One recent method of injecting chaos into BAM is through the output function on recall. This can be accomplish by increasing the transmission parameter δ in equation (2.13) to the chaotic region of its bifurcation diagram. Chartier et al [2, 11] were able to demonstrate that even with chaos the output of each unit can be bounded to a region of state space. Furthermore higher values of δ allow unconstrained wandering of state space.

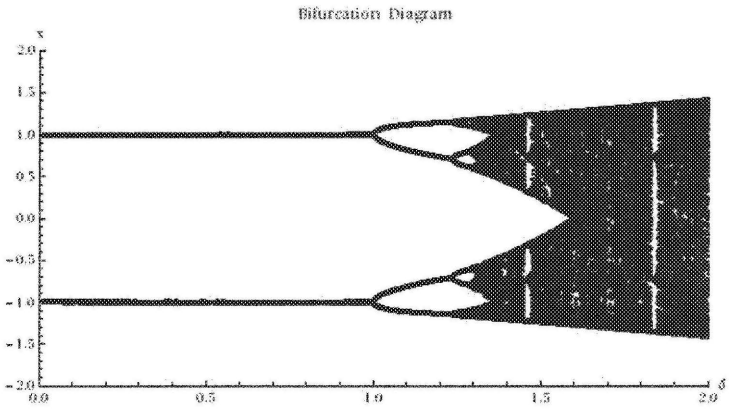

Figure 2.6: Bifurcation diagram for new output function with transmission parameter δ [2].

These results shows, that chaos can be used as a searching progress where the transmission parameter is set to unconstrained chaos region (e.g., $\delta = 1.9$ in Fig. 2.6) and reduced as the network approaches a region of state space which contains the desired attractor (e.g., $\delta = 0.1$). This idea further reinforces concepts from psychology and neuroscience that recall is not a one shot all or nothing process. But a progressive process that also needs to be taken into account by our models. However, we should note that long transient phase would not be ideal for information processing especially in biological systems since the agent needs to constantly react to its environment that is continually

changing.

2.4 Reinforcement Learning

Reinforcement Learning is the paradigm of determining how to act in situations through trial and error. In contrast to supervised learning in which a teacher has to present matching input-output pairs to the agent to learn, and unsupervised learning in which the agent only receives input signals and is asked to make sense of the data.

Reinforcement Learning (RL) works by placing an agent into some unknown dynamical environment, in which it can perform a set of actions each of which will return a certain reward in the next state. This is shown in Figure 2.7. Here reward is used for both positive and negative reinforcements.

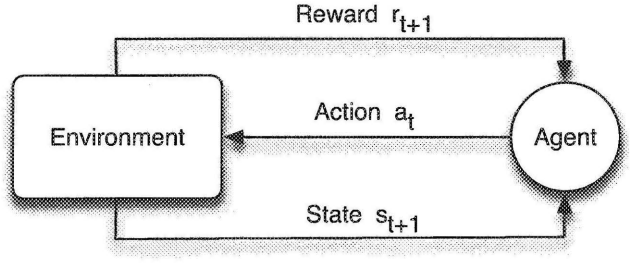

Figure 2.7: Interaction between an agent and its environment.

The goal of RL is to find a policy that maximizes the reward in the long term. A policy can be thought of as a mapping of actions to take in any particular state. Such as pressing the elevator button corresponding to the upper floor while trying to get home. In addition, a policy is optimal, if from all starting states in the environment the reward is maximized. A policy that does not change over time is called a stationary and inversely a policy that does change over time is non-stationary. However, for our main focus we will only consider stationary policies. Going back to the recent example, the optimal action to take in the elevator would be to press the button corresponding to the floor you live on to minimize the amount of time it takes to get home. It is noted that maximizing the reward can be perceived as minimizing some punishment.

It is often the case that RL problems are framed as a Markov Decision Process (MDP).

By using MDP, each problem can be expressed in a uniform manner and more generalize techniques can be develop. MDP is a well-known theory on how to make decisions in complex environment when outcomes are partially random and partially under control of the agent [39]. Any MDP can be represented as a tuple (S, T, A, R) where S is the set of states, A is the set of actions, $R : S \times A \rightarrow \mathbb{R}$ is the reward function of performing an action in a certain state and $T : S \times A \times S \rightarrow [0, 1]$ is the state transition function that is the probability of being in the successor state after performing an action in the current state.

A MDP assumes the markov property holds in the environment, that is the successor state only depends on the current state and action taken. In addition there always exists an optimal policy for MDPs. There are two main ways to find policies, direct or indirect, however we will only consider indirect methods. Direct methods entails representing the policy by a set of adjustable parameters which are changed to get a better policy. But since no gradient information is available for discrete problems samples of the return are used, e.g., variations of gradient methods, simulated annealing, evolutionary algorithms, etc. Indirect methods are lesser computationally in the sense that for each sample the estimates are shared by many policies and adjusted accordingly. These methods work by using states values or action values. For the state values we seek to place a numeric value on the desirability of a state when following the current policy and action values is the desirability of choosing an action in the current state and follow the current policy afterwards.

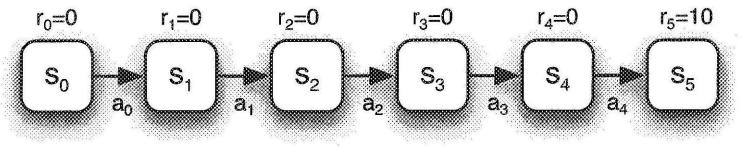

Figure 2.8: Delayed reward for in a six state environment.

Reinforcement learning problems can be categorized into two groups based on difficulty. The first of which were named immediate reinforcement problems. In these problems the optimal action is the action with maximum reward for each state. However, the

more general reinforcement problem considers delayed rewards as shown in Figure 2.8. In other words, a significant reward could be only presented after a long sequence of states. Where the task is to give credit to the actions responsible for the reward. This is usually term the credit assignment problem.

In general reinforcement problems the transition function, sometimes called the model, and the reward function are usually unknown. The agent must decide what information it needs to store. Agents that solve the RL problems without building a model are called modelfree methods in contrast to agents using experience to build a model. Three common approaches to solve the RL problem are by using Dynamic Programming, Monte Carlo simulations or Temporal Difference methods [40]. All these methods are derived from the Bellman equations:

$$V^{\pi}(s) = R(s) + \gamma \sum_{s'} T(s, \pi(s), s') V^{\pi}(s') \tag{2.15}$$

Which indicates that the value of state s while following policy π is the reward $R(s)$ plus the value of all successor states condition on the probability of reaching the said state by taking action $\pi(s)$. $\pi(s)$ the action taken in state s by policy π. Action selections are usually simple computations based on the value functions. A common deterministic approach is to always choose the action with the greatest action value. But most often, it is suboptimal to always choose the greedy action. Instead a better approach called ϵ-greedy is to choose the greedy action with probability $1 - \epsilon$ and all other actions ϵ of the time. [40]

Dynamic Programming (DP) methods are derived from control systems and assume a model of the environment. Two common algorithms to find solutions are Value Iteration [41] and Policy Iteration [42]. DP methods are optimal in the sense that it does as best as possible. However, they are not feasible for larger state space but are ideal to benchmark other algorithms against such as Monte Carlo and Temporal Difference methods. The reason DP is not feasible for large state space is that it updates all state values simultaneously regardless of the current state and the action taken. In addition the transition probabilities have to be completely specified that requires order $|S| \times |A|$

entries. For a state space with $n = 10^{120}$ states, e.g., chess, that entails trying to solve n equations with n unknowns. For any reasonable problem, it becomes impractical since it cannot be computed before hand or too large to store and maintain. Instead, it would be extremely efficient to only update the states that were visited after each transition. That's the idea behind Monte Carlo and Temporal Difference sampling methods.

The Monte Carlo (MC) method does not assume complete knowledge of the environment. Instead it gains knowledge from sample experience or sequence of states, actions, and rewards. Because of this MC methods only update their states at the end of an episode where an episode is defined as a sequence of experience that ends in a terminal state. Thus, to use MC methods we must guarantee that all experiences eventually terminate. However, the advantage of this approach is that it does not rely on previous state estimates for updating, also known as boosting, it is able to perform much better than other methods in non-markovian environments [40]. These methods are ideal for situations where real work experience is costly but simulations are cheap [43].

2.4.1 Q-Learning

In contrast to MC methods, Temporal Difference (TD) methods update the state or action values after each step instead of after an entire episode. These methods are quick to adapt to changes in the environment and also do not rely on a model. Instead, they take advantage of boosting by using previous estimates as a basis for making new estimates. One such off-policy method is Q-learning developed by Watkins [5]. The convergent proof was later provided by [44]. The general idea behind Q-learning is to store action values instead of states. This eliminate, the requirement to construct a model of the environment when selecting actions. A greedy policy can be used to select an action a that maximizes $\max_a Q(s, a)$. So we can update action values by:

$$Q_{t+1}(s, a) \leftarrow Q_t(s, a) + \alpha[r + \gamma \max_{a'} Q_t(s', a') - Q_t(s, a)] \tag{2.16}$$

Action values are updated by the TD rule that moves our estimate of $Q(s, a)$ closer to $r + \gamma Q(s', a')$ which is assumed to be a better estimate by the fact that it includes reward

from an experience. Furthermore, s' and a' are the successor state and next action. With an appropriately small value for α Q-learning has been show to converge to the optimal value function Q^* with probability 1. A full listing of the Q-learning algorithm can be found in Table 2.1.

Table 2.1: Q-learning algorithm.

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
    Initialize s
    Repeat (for each step of episode):
        Choose a from s using policy derived from Q(e.g., ϵ-greedy)
        Take action a, observe r, s'
        $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$
until s is terminal

2.4.2 Eligibility Trace

TD methods, (referred to as TD(0)), are quite an improvement over DP methods in terms of computational time. But only one state is updated at a time and the propagation of state values when a significant reward encountered can be quite slow [40]. On average it takes n episodes to propagate a terminal state value back n states if the same path is taken. In contrast a MC method would update all states visited after each episode. Eligibility traces, pioneered by Sutton [6, 7, 8] are a comprise between these two extremes. The basic idea is to assign a number (trace) to each state when visited which determines its eligibility to be updated when the agent receives a reward in the future. The most recent states will have higher traces and therefore share more responsibility for any immediate reward. This can be seen as a mechanism to help solve the credit assignment problem.

$$e_t(s, a) = \begin{cases} \gamma\lambda e_{t-1}(s, a) + 1 & \text{if } s = s_t \text{ and } a = a_t \\ \gamma\lambda e_{t-1}(s, a) & \text{otherwise} \end{cases} \tag{2.17}$$

In (2.17) the parameter λ controls how fast previous traces falls off from the current

state. By adjusting the parameter λ towards 1 we move closer to a pure MC method TD(1) and inversely towards 0 we move back to the original TD(0) learning. In fact it was shown by [40] that better performance are attained with intermediate values for $0 \leq \lambda \leq 1$. To distinguish tracing methods from non-tracing methods we will refer to them as TD(λ) methods and in particular Q(λ).

Remember Q-learning maintains two policies, the one it follows (behavioural) and the one which is being learned (estimated). Since the behavioural policy will sometimes take actions that are sub-optimal with respect to the estimated policy it may not be best to indiscriminately apply eligibility traces. A non greedy action might take you to a completely different set of states. In other words, we are using exploratory actions in estimating the value of following the greedy policy! As a result if you mark every action value as eligible, you backup the effect of non-greedy actions. Watkin's [5] proposed instead to look ahead only one step pass the next exploratory action. However because of the frequency of non greedy actions at the start learning will be slow from terminating traces.

Peng's Q(λ) [45] addresses this by having no distinction between exploratory and greedy actions. It tries to balance the two by updating the policy early on using exploratory actions and later to greedy actions. The only disadvantage of this method is its complexity to implement. There is a third approach which we will be using that just applies the regular trace to TD learning called *naive* Q(λ). Although backing up exploratory actions seems bad, it has shown good performance [46] in comparison to Watkin's Q(λ) and Peng's Q(λ).

In the previous section the traces defined, now call accumulating traces can be improve if you consider what happens when a state is repeatedly visited by an agent. The eligibility for that trace becomes greater than 1 and this would cause the agent to take more responsibility than required for the reward and become a problem for convergence. To overcome this we can consider replacing the trace (replacing traces [8]) instead of incrementing by one. Hence we do:

$$e_t(s,a) = \begin{cases} 1 & \text{if } s = s_t \text{ and } a = a_t \\ 0 & \text{if } s = s_t \text{ and } a \neq a_t \\ \gamma\lambda e_{t-1}(s,a) & \text{otherwise} \end{cases} \tag{2.18}$$

Thus the update rule for Q(λ) for all states turns out to be:

$$Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \alpha\Delta_t e_t(s,a) \tag{2.19}$$

and the error is calculated only for the current state:

$$\Delta_t = r_{t+1} + \gamma \max_{a'} Q_t(s_{t+1}, a') - Q_t(s_t, a_t) \tag{2.20}$$

These eligibility values are then used in the full update rule to get Q(λ) algorithm which is listed in Table 2.2.

Table 2.2: Q(λ) Algorithm with replacing traces.

Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
    Initialize s
    Repeat (for each step of episode):
        Choose a from s using policy derived from Q(e.g., ϵ-greedy)
        Take action a, observe r, s'
        $\delta \leftarrow r + \gamma \max_{a'} Q(s',a') - Q(s,a)$
        $e(s,a) \leftarrow 1$
        For all s, a:
            $Q(s,a) \leftarrow Q(s,a) + \alpha\gamma e(s,a)$
            $e(s,a) \leftarrow \begin{cases} 0 & \text{if } s = s_t \text{ and } a \neq a_t \\ \gamma\lambda e_{t-1}(s,a) & \text{otherwise} \end{cases}$
    $s \leftarrow s'$
until s is terminal

2.5 Exploration and Exploitation

In RL problems we are not given the model nor the reward function we must explore the environment to learn its structure. However this introduces a problem since RL task are usually online we are faced with a dilemma. Whether to exploit knowledge we already have or be optimistic and keep exploring for better potential rewards. This is a problem face not just by action value agents but in addition those that build a model as well. For taking the optimal action with respect to the agent's model of the environment may not be optimal at all because the agent's model could be incomplete or outdated.

A reasonable approach is to have an overly optimistic start by over estimating the true state values. Using standard methods the agent will explore optimistically states only to learn its true value and possibility find the optimal policy. This heuristic led to the development of a model base learning algorithm call R-MAX [47]. Additionally, we could use an exploratory policy such as an ϵ-greedy for the behaviour policy while learning the optimal policy independent of the policy being followed. A provable optimal method of making the trade off between exploration and exploitation is based on the idea of computing Gittins indices [48]. Unfortunately, so far the method has not been extendable to the general reinforcement problem.

2.6 Associative Reinforcement

A sensible way to create an associative system is to incorporate Reinforcement Learning on top of an associative memory model. A major system was developed by Zhu [12] using the PALM network. A plam network [49] functions like a BAM with weight update rule:

$$\mathbf{W} = \bigvee_{n=1}^{N} [y_n x_n^T] \tag{2.21}$$

where the operator $\vee$ is boolean OR. In other words, each weight connection is limited to either 0 or 1. A single layer feed-forward PALM network $\mathbf{W}_{critic}$ was used to store the critic (state-to-value) and $\mathbf{W}_{act}$ the policy (state-to-action) with a modified online learning rule:

$$\mathbf{W}_{act} \leftarrow \mathbf{W}_{act} + \Delta\mathbf{W}_{act} = \mathbf{W}_{act} + (r + \gamma Q_{t+1} - Q_t) \cdot s_t \tag{2.22}$$

$$\mathbf{W}_{critic} \leftarrow \mathbf{W}_{critic} + \frac{\alpha \Delta \mathbf{W}_{act} \mathbf{W}_{critic}}{\sum_{i,j} \mathbf{W}_{critic}[i,j]} \tag{2.23}$$

given that $Q_t = \mathbf{W}_{critic} \cdot s_t$ and $Q_{t+1} = \mathbf{W}_{critic} \cdot s_{t+1}$. Initially $\mathbf{W}_{act}$ is set to random values and $\mathbf{W}_{critic}$ is given small positive values, e.g., 0.1. The policy affects the state values by which actions are selected so the state could be updated and the state value affects the policy by influencing which action is chosen. Furthermore, on recall the system uses the rule below to extract an appropriate action while in some state s_t at time t:

$$a_t = f(\mathbf{W}_{act} \cdot s_t - \theta) \tag{2.24}$$

The action component uses a 'k winners take all rule' where only k bits in the output are activated ($a_t = 1$) as the action vector. If more than k units are active a random subset of k are selected. This is believed to be the source of exploration needed for optimal learning. This encoding mechanism forces the actions to self organize. The threshold parameter θ is adaptively set to make sure only the top k bits are activated. Their system was reported to have good convergence and generalization with $k = 1$ winner take all in comparison to Q-learning on a 15-by-15, deterministic and stationary grid world task. The downside of this model is that it is not based on dynamical system principles and hence cannot be used as a model of learning in the brain.

However a similar idea has been attempted in the literature recently by Chartier [26]. This approach requires adding an asymmetric parameter h to the output function from equation (2.13):

$$x_{i+1} = h + (\delta + 1)a_i - \delta a_i^3 \tag{2.25}$$

The justification of this parameter is to bias the search space in the direction of the parameter h such that attractors near that region gains a larger radii during the recall process. Results show that recall performance was boosted for patterns near the region of parameter h (i.e., patterns similar to h). A bias value was generated for each input the

network associated incorrectly by a secondary BAM network. Chartier was able to use a collection of BAMs to learn a non-linearly separable task, the XOR problem, and solve immediate reinforcement problems. This was thought to be impossible using Hebbian derived learning rules. However this approach required a secondary BAM that is told the correct actions which in general is a form of supervised learning. The goal of our system is to extend the abilities of associative memories model to more general reinforcement problems by considering temporal sequences with delayed rewards.

Chapter 3

Proposed Model

The previous methods discussed in section 2.6 either used gradient based function approximators to store the value function or supervised learning by giving the correct actions to the network. We propose a method based on Hebbian learning that takes advantage of AM networks unique ability of pattern completion. In this section, we will outline the theory behind our approach, the network architecture and implemented details. Two sample problems from the literature will also be introduced and used as a benchmark for our system in the following chapter.

3.1 Theory

The key insight was to represent the value function by the energy function of an autoassociative memory network. By using context units, we were able to store state-action pairs in the network and the energy of these states corresponds to the negation of action values. From here on, we will adapt the terminology of using x for the state the network is in and reserve s for the state received from the environment which the network associates with a reward.

For now, without mentioning the detail of how the network would be trained to reach these very specific energy levels, we will briefly explain why this concept is sound. First, consider a fully trained network with the correct state-action pairs and action values (energy levels). Such as the 1-dimension network shown in Figure 3.1 that has a return

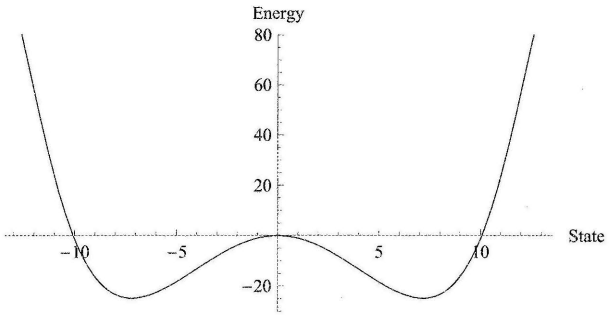

Figure 3.1: Energy function of 1 dimensional state system with -1 stored as return for the states +1/-1.

of -1 stored at states $+1$ and -1. For some particular state x the network might be in, a policy must be present that chooses an action. This could work by attempting to complete the pattern for the state-action pair (correct action vector is not available). But from this error correcting procedure the network converges to the action with the lowest energy which so happens to correspond to the action value with the highest return. It is by this reasoning the energy level is the negation of the action's return value.

3.2 Design

Our system ($VPTF$) is composed of four inter-connected components: value function (V), policy (P), eligibility trace (T) and feature extraction (F). The components work together by first receiving $\mathbf{s}_t$ from the environment and the relevant features are extracted as shown in Figure 3.2. The current state is passed onto the policy where an action is taken that transfers the agent to the next state $\mathbf{s}_{t+1}$ which is filtered to s_{t+1} and the greedy action recommended is stored. These along with the current reward r_{t+1} are then used to update the action values of the old action and trace back in time to update all previous experience. All the components will be discussed in more detail through the next 4 subsections.

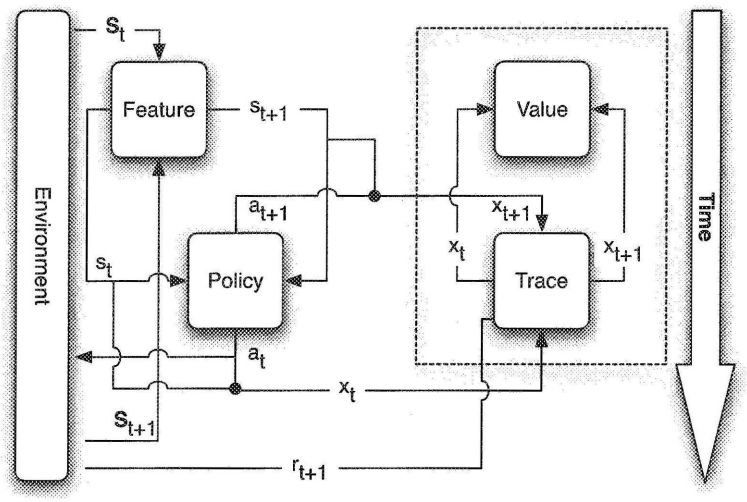

Figure 3.2: Overview of the interaction between the different components.

3.2.1 Value Function

Network V is the core component of the system and will be used for associating state-action pairs to action values as shown in Figure 3.2. Network V will be represented by a single autoassociative network. Here, the environment state units are used as context units for the current action to disambiguate between correct actions in different states.

On each iteration the network update its estimate of the optimal value function by changing the weights according to the Q-learning update rule. But first the old action values are retrieved as the negation of the energy from the current network state $x_0 = sa$.

$$Q(s,a) = -E(\mathbf{V}, sa) \tag{3.1}$$

$$Q(s',a') = \max_{a'} -E(\mathbf{V}, s'a') \tag{3.2}$$

$$Q(s,a) = Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)] \tag{3.3}$$

where it is understood that we are in state s and took action a then transitioned to state s' and the greedy action is a'. After the new estimate of $Q(s,a)$ is used to update the weights as:

$$\mathbf{V} = \mathbf{V} + \eta[-Q(s,a) - E(\mathbf{V}, x_0)] \cdot [x_0 x_0^T - x_t x_t^T] \tag{3.4}$$

It should be noted this learning rule is a generalization of the previous learning rule and collapses back to equation (2.12) when $E(\mathbf{V}, x_0) = -Q(s,a) - 1$. However that will not normally be the case and more generally there are two fixed-points. The original at $x_0 x_0^T = x_t x_t^T$ and the other when $E(\mathbf{V}, x_0) = -Q(s,a)$.

Since roughly speaking the energy level of a bipolar pattern that is a fixed-point can be approximated by:

$$E(x) = \frac{1}{2}[x^T y - (\delta + 1)x^T W y + \frac{1}{2}\delta x^T (Wy)^3 + C] \tag{3.5}$$

$$= \frac{1}{2}[n - (\delta + 1)n + \frac{1}{2}\delta x^T x^3] \tag{3.6}$$

$$= \frac{1}{2}[n - n - \delta n + \frac{1}{2}\delta n] \tag{3.7}$$

$$= -\frac{\delta n}{4} \tag{3.8}$$

which is typically around $\frac{-\delta n}{4}$ for a n-dimensional network. So we should make sure our return values are mapped to a higher interval to avoid interference with the first fixed-point in equation (3.4). We also assume for the energy function that the constant $C = 0$. For consistency with the theory of Reinforcement Learning theory we will consider the negation of the energy function instead:

$$e(x) = -E(x) \tag{3.9}$$

The effect this has on the shape of the energy function can be readily seen in Figure 3.3 in comparison with a conventional trained network for the patterns $(1,1)$ and $(1,-1)$ at energy levels -1.9 and -1.5. Note its structure compare to a conventional network trained with the same fixed-points. The original $E(x)$ is scaled on the x_1 and x_2 axis until the desired energy level is reached at the bipolar points. This causes the original fixed-points to be shifted outwards to new coordinates.

3.2.2 Policy

The second component P implements the policy the network follows by using the structure of the value function in the V network. By iterating the network V from a known initial

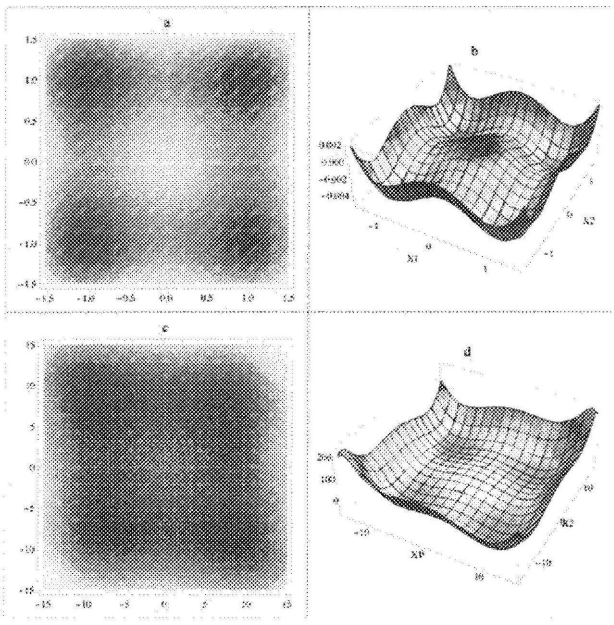

Figure 3.3: Original associative memory energy function of 2 dimension network with fixed-points at the corners of the cube. Again, a network with $(1,1)$ and $(-1,-1)$ stored at -1.9 energy level and $(1,-1)$ and $(-1,1)$ at -1.5 energy level.

network state $x_t = sa_0$ where a_0 can be some arbitrarily chosen action or $\vec{0}$ and s is the current state from the environment. In addition having s clamped to s_t, the network will traverse the landscape of the energy function by following the path that leads to a local minimum but being restricted to the action subspace. An incremental approximation to the next nearest action a_{k+1} with lesser energy is:

$$s_t a_{k+1} = f(\mathbf{V} \cdot s_t a_k) = (\delta + 1)\mathbf{V} s_t a_k - \frac{1}{2}\delta(\mathbf{V} s_t a_k)^3 \tag{3.10}$$

In that region, the action that minimizes the energy function will corresponds to the action with maximum return. This is analogous to having an attractor ruin in high dimensional space but orbits being limited to a small manifold. However outside changes (new state from the environment as a consequence of choosing an action) will cause the network to suddenty transition to a new stable manifold. We can imagine these manifolds are represent by clouds in Figure 3.4 where different actions taken in a state corresponds to following an orbit to another state. It is in fact the agent's organization of the environment.

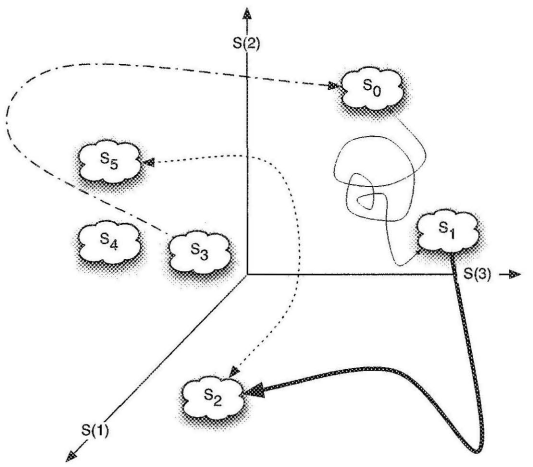

Figure 3.4: States represented in high dimension and actions as orbit between states.

Fortunately the exact state values are not required to extract the optimal policy. Usually the optimal policy is available before the action values fully converge to the optimal value function so we will make use of this inside our policy.

A second step is to take advantage of chaos as a searching process. In an ϵ-greedy policy we usually select a random action with probability ϵ by using a stochastic process to sample from this distribution. We propose to use an analogous mechanism with chaos to sample from the space of actions. Consider a network $\mathbf{V}_a$ that is trained in advance with all the valid actions the agent can perform. If the network is probed with an arbitrary vector it will converge to one of these actions or its complement after a sequence of iterations. However if we initially set the transmission parameter δ to an unconstrained chaotic region (e.g., $\delta = 2$) of the network's bifurcation diagram. This will cause the network to arbitrarily cross boundary axes and transition between patterns unpredictably. It is apparent from Figure 3.5 where we show a 2 dimensional network with an initial state of $(0.4, 0.5)$, $\delta = 2$, and weights:

$$\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$$

5000 iterations are plotted on the plane. It can be compared to points generated randomly in the same interval uniformly where there is approximately equal amount in each quad-

rant. We will initially set the transmission parameter δ to high values which corresponds to unconstrained chaotic region and reduce the value as the agent becomes more confident in its value function where the optimal action we will slowly move to a more stable regions. The motivation behind this is to use a naturally occurring process of the network to our advantage as it can be seen as a very plausible technique for biological models. Furthermore recalling as we have explained is not a one shot process but a slow iterative process of reconstruction. If the two ideas mention above are combined then the state should be stored for the chaotic orbit to be resumed later. A simple method to combine these actions is a linear sum based on the parameter ϵ, i.e., $a \leftarrow (1-\epsilon) \cdot a_{greedy} + \epsilon \cdot a_{explore}$.

A consequence of using the associative memory model is the inverse of states and actions are also stored with the same energy level. That means on recall there is a possibility of retrieving the inverse action. On recall, our policy will recommend an action a, to determine the correct action we take the action a_i that has the greatest absolute correlation of all actions. i.e., $\max_i |corr(a, a_i)|$. Actions will generally not interfere with each other but the correlation between actions determine how easy it would be for the network to transition to a near by action on recall. So for neutrality each actions should be equal distance (bitwise) from all other actions. One simple encoding of this is the 1-of-d choice. For a d dimensional vector only one component is active while the rest is inactive, e.g., $(-1, -1, +1, -1)$.

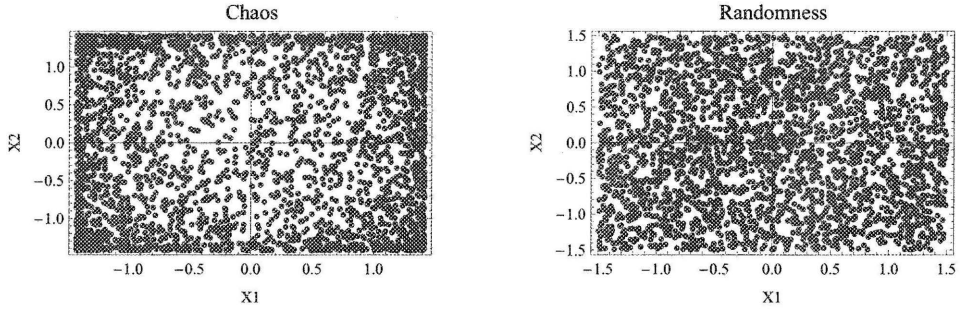

Figure 3.5: Two dimensional network with transmission parameter (δ) set to 2.0 and weights set to the 2-by-2 identity matrix with starting states of $(0.4, 0.5)$. In contrast with randomness on the interval $[-1.5, 1.5]$.

3.2.3 Eligibility Trace

The third network T stores a trace of the most recent states and actions taken. Similar to an eligibility trace any reward receive are propagated back to previous states. The attraction of this component is that it allows the system to more quickly propagate changes backwards by informing earlier state-action pairs of their responsibility. In fact the mechanism is a technique to help solve the credit assignment problem discussed earlier.

However only a fraction of states that are recently visited will have trace of any significance since the rest will have near zero values. Instead, we will only keep track of states that were visited recently. The list of recently visited states can be viewed as a limit cycle at the BAM network level. So the network T is represented as a BAM and for each episode the initial state is stored as a fixed-point and inductively the current state is linked to the previous which forms a chain as shown in Figure 3.6. The initial state is made a fixed-point to place a terminal state in the chain.

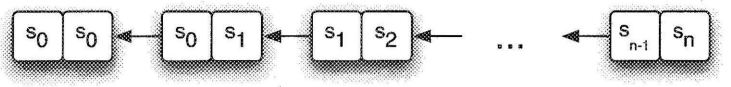

Figure 3.6: BAM network storing successive states iteratively.

The propagation of rewards backwards would be done after each time step.

$$Q_{t'}(s,a) \leftarrow Q_{t'}(s,a) + \alpha \Delta_t e_{t'}(s,a) \text{ for all } t' \leq t+1 \tag{3.11}$$

This update is similar to a regular Q-learning update of a state but here the error is stored in a term Δ and used in all recently visited action values:

$$\Delta_t = r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t) \tag{3.12}$$

Also note the $Q_t(s,a)$ values are updated in network V.

3.2.4 Feature Extraction

The last network F was inspired by the use of feature extracting abilities of the BAM. We want to compress the state space for large problems into smaller sizes. This is a desirable property of any intelligent system since in many problems the state space is intractable to enumerate and furthermore only a fraction of states are necessary to visit in order to find a near optimal policy.

However it should be noted that this method builds generic states as attractors to represent common states the agent encounters. In particular we would not expect rarely encountered states to be represented well. In fact, we expect our network to filter out and map the most common or important states from high dimensional to a low dimensional subspace.

That means the previous networks mention (i.e., networks V, P, and T) will only see a limited set of states from the original problem in a lower dimension. This brings forth another advantage of function approximators (FA), the ability to generalize from states, closer states will have similar state values. In fact with this extra layer our system has two levels of approximators. That is the original BAM by itself can be used as a FA with fix-points defining areas of interest in state space and any near by points (basin of attractor) would naturally be attracted to the fixed-point as this is one of the fundamental capabilities of associative memory networks. However it is known that standard RL algorithms, in particular QL has had problems converging when using FA compare to tabular representations and have only been proven to converge on linear approximators and usually diverge on nonlinear approximators without extensive tweaking [50, 51].

Without much modifications as illustrated in Figure 3.2 the incoming state $\mathbf{s}_t$ from the environment is trained on the network and its extracted features s_t are passed onto the other system as a lower dimension state. So we will use the following equations for training weights $\mathbf{F_w}$ and $\mathbf{F_v}$:

$$\mathbf{F}_w(k+1) \leftarrow \mathbf{F}_w(k) + \eta(y_0 - y_1)(x_0 + x_1)^T \tag{3.13}$$

$$\mathbf{F}_v(k+1) \leftarrow \mathbf{F}_v(k) + \eta(x_0 - x_1)(y_0 + y_1)^T \tag{3.14}$$

with x_0 as the initial state $\mathbf{s}_t$, $y_0 = kwta(\mathbf{F}_w x_0, k)$, $x_1 = \mathbf{F}_v y_0$ and $y_1 = \mathbf{F}_w x_1$. By deriving y_0 we used the function $kwta$ to limit the amount of units that can fire and control the sparseness of encoding, i.e., pick the top k units to fire. On recall, we are given a state $\mathbf{s}_t$ and one trial through the network we derive s_t:

$$s_t = f(\mathbf{W}f(\mathbf{V}f(\mathbf{W}\mathbf{s}_t))) \tag{3.15}$$

that is later use throughout the network.

3.3 Implementation

Our algorithm is based off a common RL solution called Q-learning. The algorithm was implemented in the Mathematica programming environment. The reason being it was very expressive and supported many required features such as matrix operations and graphing. The source code is also available in Appendix A and the pseudocode of our algorithms are in Tables 3.3, 3.1, and 3.2.

Initialize $\mathbf{V}_\alpha, a_{explore}$ in trial 0
Initialize $\mathbf{V}, s, \epsilon$
$a_{explore} \leftarrow f(\mathbf{V}_\alpha \cdot a_{explore}; \delta = 2.0)$
$a_{greedy} \leftarrow f^{(10)}(\mathbf{V} \cdot \vec{s0})$
$a_\beta \leftarrow (1-\epsilon) \cdot a_{greedy} + \epsilon \cdot a_{explore}$
Let $A_i = |cov(a_\beta, a_i)|$ for all $i = 1, 2, ...|a|$

$$j \leftarrow \begin{cases} \max_j A_j & \text{if RandomReal}(0,1) \geq \epsilon \\ \text{RandomInteger}(1, |a|) & \text{otherwise} \end{cases}$$

Return a_j

Table 3.1: Dynamic policy.

An issue encountered during implementation was that it was problematic finding the chaotic region for the transmission parameter since the fixed-points has been moved out of the usual interval. Instead we used a fixed network $\mathbf{V}_\alpha$ with the original action vectors trained and conditioned on it's chaotic region to regenerate exploratory actions. We used

the covariance of the recalled action against all actions to infer which action is closer to the attractor during action selection for the dynamic policy. That is the action chosen by our agent.

Initialize $\mathbf{V}, s, \epsilon$
Let $A_i = e(\mathbf{V}, sa_i)$ for all $i = 1, 2, ...|a|$

$$j \leftarrow \begin{cases} \max_j A_j & \text{if RandomReal}(0,1) \geq \epsilon \\ \text{RandomInteger}(1, |a|) & \text{otherwise} \end{cases}$$

Return a_j

Table 3.2: Greedy policy.

For practical reason the trace depth was set to be $\frac{1}{3}$ of the networks dimension. Also, to avoid infinitely updates by progressing backward within a limit cycle a list of previous updates was maintain and only unique state-action pairs were updated.

Initialize $\mathbf{V}, \mathbf{C}_w, \mathbf{C}_v$ = all weights 0
Initialize $\mathbf{F}_w, \mathbf{F}_v$ to small random values
Set parameters $\alpha, \gamma, \epsilon, \lambda, \eta, \delta$
Repeat (for each episode):
    Initialize $\mathbf{E}$ = all weights 0
    $\mathbf{s_t} \leftarrow$ state from environment
    Use $\mathbf{s}_t$ to train $\mathbf{F}_w, \mathbf{F}_v$
    $s_t \leftarrow$ recall $\mathbf{s}_t$ from $\mathbf{F}_w, \mathbf{F}_v$
    Repeat (for each step of episode):
        $a_t \leftarrow$ action recommend by policy in s_t (i.e., ϵ-greedy or $\mathbf{P}$-greedy)
        If first step: associate sa with sa using $\mathbf{C}_w, \mathbf{C}_v$
        Take action a_t, observe r_{t+1}, $\mathbf{s}_{t+1}$
        $s_{t+1} \leftarrow$ recall $\mathbf{s}_{t+1}$ from $\mathbf{F}_w, \mathbf{F}_v$
        $a_{t+1} \leftarrow$ action recommend by policy in s_{t+1} (i.e., ϵ-greedy or $\mathbf{P}$-greedy)
        $q_t \leftarrow energy(\mathbf{V}, s_t a_t)$
        $q_{t+1} \leftarrow energy(\mathbf{V}, s_{t+1} a_{t+1})$
        $\Delta \leftarrow r + \gamma q_{t+1} - q_t$
        Associate $s_{t+1}a_{t+1}$ with $s_t a_t$ using $\mathbf{C}_w, \mathbf{C}_v$
        Let $x_t = s_t a_t$
        $\mathbf{E} = trainAuto(\mathbf{E}, x_t, 1)$
        Repeat (for each step taken):
            $q \leftarrow energy(\mathbf{V}, x_t)$
            $e \leftarrow energy(\mathbf{E}, x_t)$
            $\mathbf{V} = trainAuto(\mathbf{V}, x_t, q + \alpha\Delta e)$
            Let u $= \begin{cases} \gamma\lambda e & \text{if } s = s_t \text{ and } a = a_t \\ 0 & \text{otherwise} \end{cases}$
            $\mathbf{E} = trainAuto(\mathbf{E}, x_t, u)$
            $x_{t-1} \leftarrow$ traverse cycle backwards using x_t and $\mathbf{C}_w$
        until cycle found
    until s is terminal
until desired number of episode

Table 3.3: Pseudocode of Associative Memory reinforcement learning algorithm.

Chapter 4

Experiments

In this section, we will support our objective with experimental results that shows our system performs well. To begin, we first demonstrate our system is capable of storing arbitrary values in the interval $[-1, 1]$ for a set of states. This region is sufficient because any other rewards interval can be mapped to this interval by scaling all rewards by the max reward appropriately. Next, we will try to experimentally determine the capacity of action values that could potentially be stored accurately with a fixed network size. Furthermore, we will demonstrate the network on a stochastic Gridworld and modified Tetris problems. For the stochastic Gridworld task, we will do a comparison with other RL algorithms and for the Tetris game we will show the performance of the system over a variety of parameter settings.

4.1 Simulations

A number of simulations will be performed to analyze the correct behaviour of our system and its subcomponents before it is tested on a full problem.

4.1.1 Value Function

To demonstrate the effectiveness of the system, we will confirm our hypothesis that the energy function is a viable means of storage for action values. The purpose of this experiment is to show the network is capable of storing the correct return values and it

System	Parameter	Value
Associative Memory	State dimension	40
	Action dimension	4
	η	10^{-4}
	δ	0.01
Reinforcement Learning	α	0.9
	γ	0.9
	ϵ	$\frac{1}{t}$

Table 4.1: Parameter settings for the Gridworld learning problem.

converges as a function of the number of iterations. The network size will be selected to be 44 dimensions with 44 bipolar states. This leads to a network load of $44/44 = 100\%$. The parameter values are set according to Table 4.1. The results can be seen in Figure 4.1.

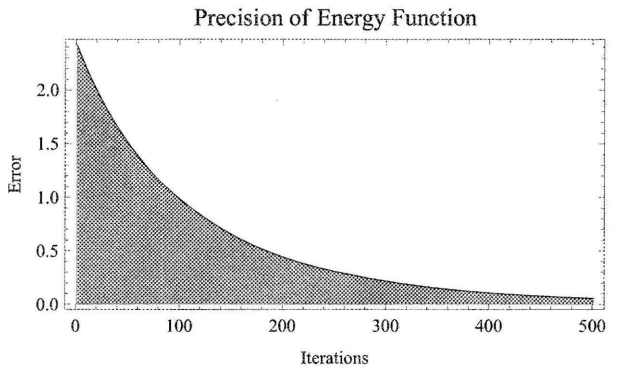

Figure 4.1: Mean squared error of setting the energy function of arbitrary bipolar states in the network to various values in the interval [-1,1].

As we would expect the RMS (root mean squared) error is reduced to almost zero after 500 iterations which is roughly 12 updates per state. These results are impressive and confirm that the network can store action values to arbitrary accuracy in the interval $[-1, 1]$. However, as was stated earlier, an n dimension network is limited to storing values only in the interval $[-n, n]$. The same test was tried for the full range of values and we saw greater error as the network needs to be distorted at greater extremes to reduce the error. For our purposes we will only consider the subinterval $[-1, 1]$ since larger intervals for rewards can be easily mapped to this range and it contains less variability for the

values being stored.

4.1.2 Action Values Capacity

In this section we will show the capacity of the network under different load factors and how this affects the error of the stored state return values. The objective behind this simulation is to justify the ratio of state values relative to the dimension of the network. Moreover, we theorize that the storage capacity (error in energy function) would be more than the commonly used load factor of 30% used in regular associative memories since we are not fully storing the state as a fixed-point. We are storing state value which is less information. In Figure 4.2 we see an increase in error and greater variance of a fixed size network of 20 dimensions storing varying number of bipolar patterns from 1 to 40. The state values were generated uniformly in the interval $[-1, 1]$. Each pattern was updated on average 50 times over 50 independent trials.

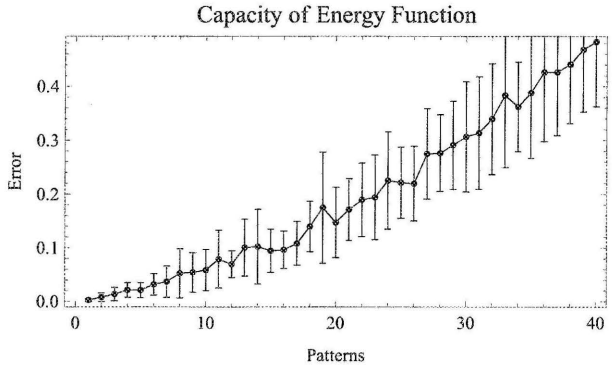

Figure 4.2: Capacity of network as a function of different state values being stored for a fixed network size in the interval [-1,1].

From the graph we can see that the rate of increase in error appears to be linear in the load factor up to 200% (40 patterns). These results shows our system is robust over varying load factors and it is unlikely that a particular threshold will degrade performance critically.

4.2 Test Problems

To show the success and generalization ability of our system we will test it on two problems. The first is a Gridworld problem and the second is a modification of the popular Nintendo game Tetris.

4.2.1 Gridworld

Our first test case is a stochastic Gridworld environment from [39]. The environment will consist of 4-by-3 cells shown in Figure 4.3. The two terminal state each give a reward of $+1$ and -1 while all other transitions are -0.02. In this world there are 4 possible actions {north, west, south, east} that correspondingly move the agent from a grid position {up, left, down, right} and remains in the same position if movement is off the grid or towards cell (2, 2). However each action has a 20% chance of failure meaning the action north will move up with 80% success and 10% chance of moving left and 10% right. Taking action west would go 80% left, 10% up and 10% down and the same for the remaining actions. The objective for the agent is to reach the positive terminal state from the start state by moving as quickly as possible to minimize the accumulating negative rewards but safely and avoid falling into the negative terminal state from a faulty action.

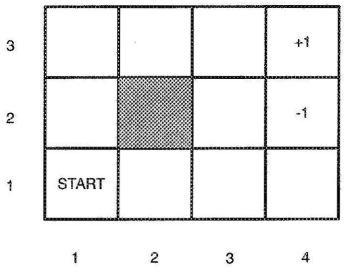

Figure 4.3: Stochastic gird world environment with an action failure rate of 20%.

For this problem we will represent actions as 4 dimensional vectors where {north=$(1,-1,-1,-$ west=$(-1,1,-1,-1)$, south=$(-1,-1,1,-1)$, east=$(-1,-1,-1,1)$}. The 11 unique states are uniformly sampled from {-1,1} to produce a 40 dimensional vector. The remaining parameters that were used can be found in Table 4.1. Since the state vectors are not based on topographical features of the states according to its 2 dimensional representa-

tion but instead were uniformly generated we will not apply the feature extraction on this problem. In addition, on such a small world performing trace would not help much.

Result

We first compare our system using the ϵ-greedy reinforcement learning policy against two standard algorithms: SARSA and Q-learning. Afterwards, we will introduce the dynamic associative policy (network P). Based on tests from 200 trials and up to 200 episodes the performance of these different systems can be seen in Figure 4.4.

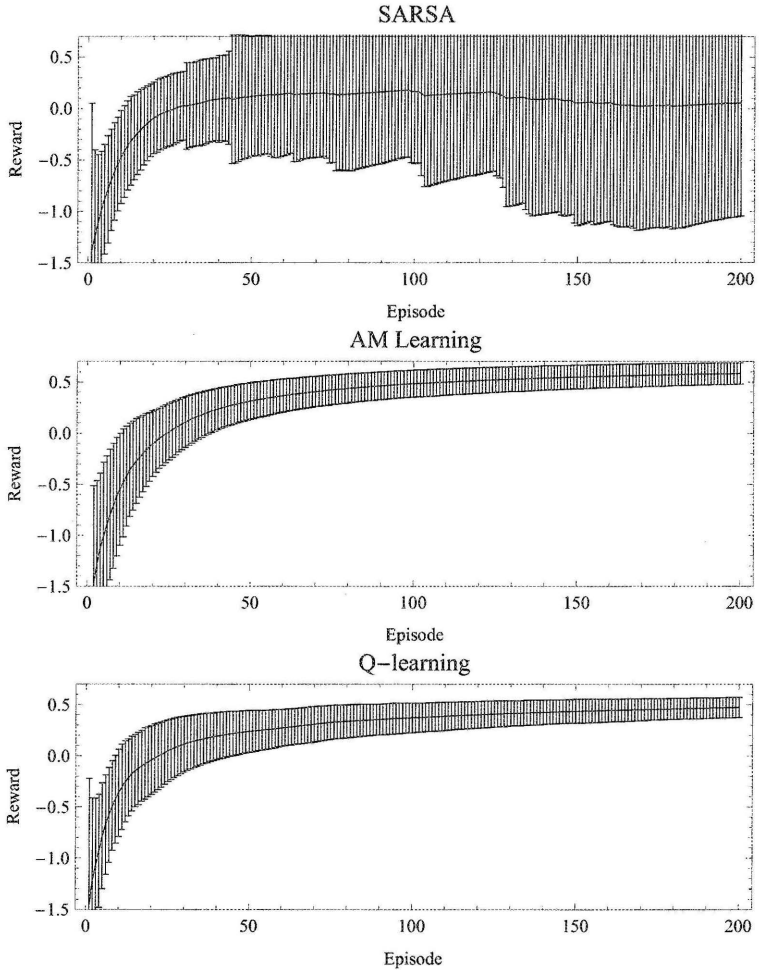

Figure 4.4: Performance from left to right of SARSA, Q-learning and the Associative Memory (ϵ-greedy) model on the stochastic grid world task measure by reward per episode over 200 trials and averaged over 200 episodes.

The plot in Figure 4.4 shows three things for us *a*) how much exploration the agent

goes through *b*) the moment the system learns the optimal policy, and *c*) the rate at which the system takes advantage of the optimal policy (exploitation). From the curves we can tell that most of the learning in the system occurs between the first episode and before the graph of the accumulative rewards curve passes zero. Since on average the expected return following a purely random policy is negative returns. After which point the system mainly exploit what it has learn for the remainder of the trial. However the rate at which it exploit this knowledge is dependent on the ϵ term that is a decreasing function of time so in the limit the policy will become entirely greedy.

For this problem we did not test the optimal policy, instead our unit of measurement was the average accumulative reward received. The justification for this choice was that it was not required to know the full optimal policy since some states has such poor expected return our agent should never go there often enough to determine the optimal action. Instead the shortest path that maximize our return would be sufficient. The middle solid line shows the mean over 200 trials and each bar is the standard deviation. We decided to show the accumulated rewards instead of rewards per episode because the episodes are not independently sampled, but are influence by pass episodes. The poor results by SARSA can be explained by trials where convergence within 200 episode has failed and the system accumulates larger negative rewards. Huge negative rewards would easily overcome any positive gains by the system. This would explain the increased rewards midway through and then a decrease below zero which is supported by a change in variance. From inspection we know QL and AM Learning has done extremely well and most likely had to have found the optimal policy consistently. This can be inferred from the plots of the small bounds on their standard deviations which happen to get smaller centered around the mean as all trials eventually converge to the optimal policy. Furthermore, the AM learning algorithm achieved accumulative reward of 0.58 compare to Q-learning at 0.47. This marks a difference of over 0.1 with a p-value of $4.5 \cdot 10^{-24}$ which is statistically significant for its scale and justifies our reasoning that AM learning was able to learn the optimal policy faster. Our explanation for the increase in performance of AM learning is the shared knowledge of state returns by distributed representation. That is, each states updates affect the estimates of other states. From this, we know the

AM system has done extremely well so for using the value function component (network V).

Now we will attempt to use the dynamic associative policy outlined earlier. In QL there are two policies, the behaviour policy and the estimated policy. For the behaviour policy, we will use a chaotic network to select an action in each state the agent must follow while the optimal action recommended by our estimated policy is found through the searching process policy described in section 3.2.2. In other words, our exploratory policy is to follow a chaotic action and the exploitive policy is to follow the recommended action at a local minimum.

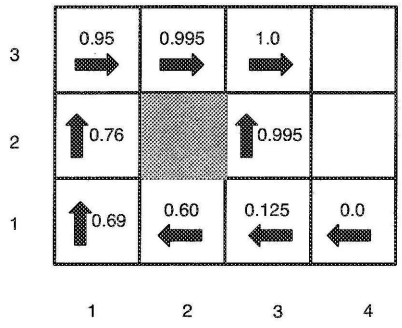

Figure 4.5: Gridworld policy when using dynamic policy with chaos averaged over 200 independent trials up to 200 episodes.

Results from the experiment can be seen in Figure 4.5. We see the proportion of times the network learned the correct action for each state over 200 trials. Returns close to the positive terminal state were learned more successfully than others, in fact the correct action for state $(4, 1)$ was never achieved in any of the trials. Even by chance this should be around 25% since there is only 4 actions. Unfortunately, as the results show the exploratory chaotic policy could not be interleaved with the greedy associative memory policy successfully. What's missing is a function analogous to ϵ-greedy that could over time transition from a pure exploratory to an exploitive policy. Therefore, it was unable to exploit knowledge gained from early exploration to its advantage and afterwards continually explores. But, although it has not maximize its reward the correct returns are being learned nevertheless a majority of the time. Additionally, it seems the network sometimes forgets what it has learned previously. It has occurred before in the

literature and is common in solutions employing function approximation. In particular by [52] coincidentally on the Tetris problem which we will mention shortly. Although the current network is not using feature extract it does indeed apply function approximation to fixed-points. In the next problem, we will continue using the ϵ-greedy policy based on the network V.

4.2.2 Tetris

Our other test case is the popular videogame Tetris. A standard game of Tetris consist of a 10-by-20 size board and it is played with 7 different tetrominos. The basic rules are that the current tetromino is placed at the top center on the board. The player must guide it downward while being able to perform basic actions. An example of a game in progress is shown in Figure 4.6 where the current tetromino is red at the top middle and the green tetromino on the side is the next piece awaiting to enter the game. Possible actions include shifting the tetromino left or right, drop it to the bottom from it's current position or rotate it by increments of 90°. When a row contains the maximum filled cells, this row is removed from play and all occupied cells above are moved down a row. However, if the current tetromino cannot be placed on the board because of occupied cells the game is over. The objective of the game is to maximize the removed rows or equivalently play indefinitely as you will need to remove rows to continue playing. A detail explanation of the rules can be found at [53]. A common modification and for our purposes is instead of letting the agent guide the tetromino, the agent will simply provide the position and rotatation from which the tetromino is dropped from the top in one action.

This is an ideal problem because of the large state space and it's difficulty in compressing. It was shown by [54] that Tetris is NP-complete. Also by playing a game using an alternative sequence of 'Z' and 'S' pieces you are guaranteed to lose using the standard board dimensions [55]. Fahey [53] conjectured that it might be possible to predict the length of a game and hence the number of completed rows by plotting the histogram of the remaining pile height after a completed row. However, as a consequence of that fact we will only consider a simplified example for faster training of the network.

In this document we will however limit our game to the 'O' tetromino piece. By

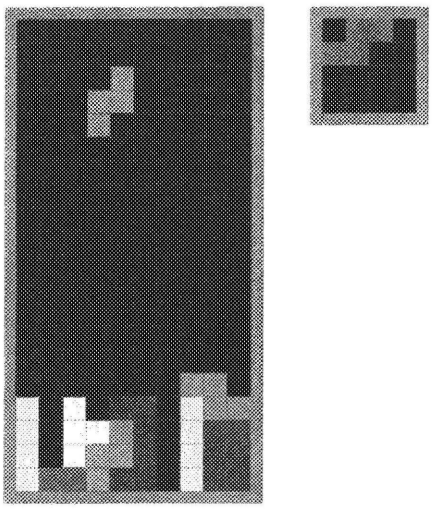

Figure 4.6: State of a game of tetris in progress.

limiting ourselves to a subset of the game we introduce an optimal strategy that an agent can use to play indefinitely. The total number of states in Tetris is approximately $2^{10 \cdot 20} \approx 10^{60}$. Although simpler than a regular game the agent must learn to navigate the large state space and focus on the relevant features of the states.

The purpose of this problem is to show our system on a challenging task where the state space is large enough that the extracted features would be quite beneficial and the length of episode sequence are sufficiently long to make eligibility trace an influence. The representation we will be using is a 200 dimension vector to represent the game's state. Since the orientation of the 'O' piece is invariant under rotation we can reduce our action space to the width of the board which in this case is 10 cells. That is sufficient because the next piece is constant and we will not consider any look ahead strategy. A simple representation was used for the action vectors. That is for action i, its corresponding 10 dimensional vector would be v such that $v_i = 1$ and $v_j = -1$ for all $j \neq i$. The parameters used for this problem are shown in Table 4.2.

Result

In this section we will show our results and analysis of our experimental results described above.

System	Parameter	Value
Associative Memory	State dimension	200
	Action dimension	10
	η_1	10^{-7}
	η_2	0.001
	δ	0.01
Reinforcement Learning	α	0.9
	γ	0.9
	ϵ	$\frac{1}{t}$
	λ	0.9

Table 4.2: Parameter settings for the Grid World learning task.

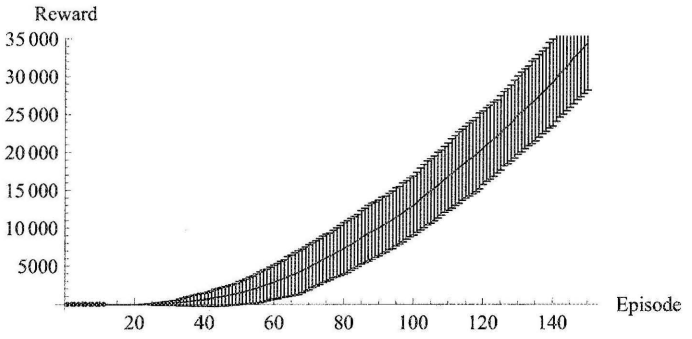

Figure 4.7: The average accumulated rows removed by training the network for 150 episodes with $\epsilon = \frac{1}{t}$ over time.

The major problem on this benchmark was that the network tends to forget what it has learned or more precisely it has over learned specific action values. Since all action-value updates effect all other actions as a consequence of the distributed representation. Our solution to overcome this problem was to set the step size parameter ϵ to decrease over time and illustrated in Figure 4.7. It could be seen that the network quickly reached a certain peak performance where it has learned the optimal policy but unexpectedly makes a poor move every so often. That actually is related to ϵ, the rate at which the network exploit knowledge it has. Since ϵ is non-zero it will eventually make a sub-optimal move by placing a block on an even position thereby guaranteeing that row will never be remove because of the combination of block being used.

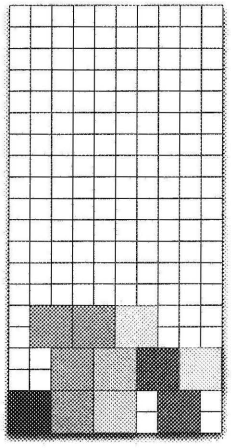

Figure 4.8: Tetris player making a sequence of bad moves that guarantee the bottom three rows can never be completed.

Furthermore, the network develops semi-optimal policy at multiple heights in Tetris, which was surprising as shown in Figure 4.8. This is related to the fact that the agent cannot recover from a bad move although there always exist an optimal action at each level to play indefinitely. On the same note, we ran the test with the same parameters but after 500 rows has been completed we change from an $\epsilon = \frac{1}{t}$ policy to the full greedy policy where $\epsilon = 0$ but still continue to update. We speculate that the network would give better performance since by that time a correct policy is known. The results can be seen in Figure 4.9. We see that in episode #44 the policy switch occurred and the network took only two more episodes to reach the stopping criteria of $50,000$ completed

rows in episode #46. At this point the network will begin to approach a fixed-point for the state values being updated and should continue playing indefinitely.

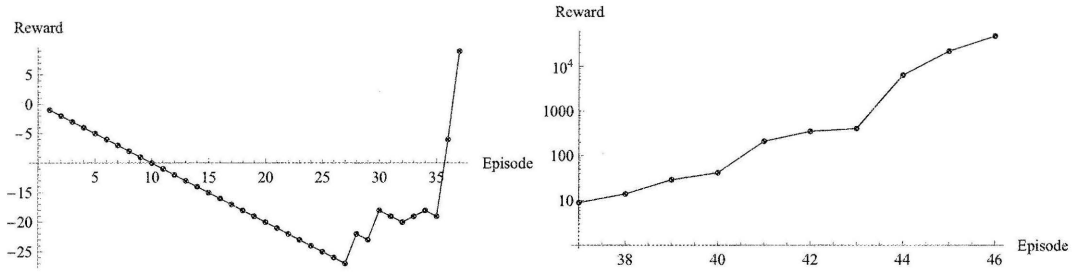

Figure 4.9: Tetris agent following the greedy policy split between episodes $1-37$ and $37-46$.

In the following experiments we will run the network with the default parameters listed in Table 4.2 for up to 500 accumulated rows removed and averaged over 20 independent trials. Each trial goes up to 200 episodes then is considered a failure if 500 completed rows was not reached. By that time we reason the correct policy should be learned. Now, consider the effect of the parameter α on the network performance. Different values of α are shown in Table 4.3. We see that better performance is achieve with lower values of α with smaller deviations from the mean at no lost of accuracy.

Statistic	$\alpha = 0.10$	$\alpha = 0.25$	$\alpha = 0.50$	$\alpha = 0.75$	$\alpha = 0.90$
Mean	44	43	58	48	48
SD	16	13	41	26	34
Accuracy	100%	100%	95%	100%	100%

Table 4.3: Performance on Tetris for various values of α using only the network V. The data was collected by noting the number of episodes required to reach an accumulated total removed row of 500 averaged over 20 independent trials.

In the next experiment we vary the parameter λ which controls the magnitude of responsibility propagated back to previous states. For example as λ approaches 1 corresponds to a Monte Carlo type of update while as λ approaches 0 is the original TD update rule. In Table 4.4 we see that for lower and higher values of λ shows better performance in the mean episode reached. This comes as a little surprising since intermediate values of λ usually perform better. Similar results can be seen for the standard deviations and all variations had no failures.

Statistic	$\lambda = 0.25$	$\lambda = 0.50$	$\lambda = 0.75$	$\lambda = 1.00$
Mean	87	90	100	83
SD	20	24	35	21
Accuracy	100%	100%	100%	100%

Table 4.4: Performance on Tetris for various values of λ using the Eligibility trace unit. The data was collected by noting the number of episodes required to reach an accumulated total removed row of 500 average over 20 independent trials

As was discussed before we can use an additional BAM to extract features from the environment states to a lower dimension before being process by the other networks. In this experiment we show the results of different dimensions mapped from the fixed 200 of Tetris in Table 4.5. Clearly the best range in this case for the Tetris problem is near 100 where in scores it has the lowest mean episode, deviation from the mean and perfect accuracy. If we go above, the extra computation introduced by feature extraction might not be worth it for the less than perfect representation of states. In the case where the dimension is too high, the network tries to extract too many unnecessary features from the original domain. As expected, at lower dimensions there is also a drop in performance. This might be explained by the network being less flexible in learning the correct features and its accuracy. But dealing with 200^2 weights to update in the original problem and 100^2 in the best state reduced version we may instead opted for 20^2 in a much smaller network with a longer training session. The lower computational requirements might be ideal in some circumstances and with network F we are able to make such trade offs.

Statistic	FE 20	FE 50	FE 100	FE 150
Mean	96	86	75	86
SD	31	41	23	28
Accuracy	95%	95%	100%	85%

Table 4.5: Performance on Tetris problem by varying the size of the projected dimension using the Feature Extraction unit. The data was collected by noting the number of episodes required to reach an accumulated total removed row of 500 average over 20 independent trials.

4.3 Conclusion

In this chapter we have demonstrated a working model of our system on various simulations and problems in particular a stochastic Gridworld problem and a modified game of Tetris. We first showed that our model can in fact store states returns using the network energy function. We later show the network is robust to varying number of returns being stored and performance gradually decrease with more states as expected. Afterwards we compare the model to two familiar reinforcement learning algorithms from the literature that is SARSA and Q-learning of which our model is derived from. The results were supportive of our thesis as our model outperformed the other methods with a p-value of $4.5 \cdot 10^{-24}$, considering our system is approximating Q-learning, and Q-learning is using tabular storage. We later showed that the dynamic policy is incapable of combining its exploratory (chaos) component and exploitative (fixed-point convergence) component as a unifying policy.

Afterwards we showed on the Tetris problem the network learning with a ϵ-greedy policy was able to achieve up to $35,000$ completed rows but we later reveal that $\epsilon = \frac{1}{t}$ was too large a probability of making a suboptimal move and was the source of many bad moves. Instead we switched to a completely greedy policy after 500 completed rows and the network exhibited unexpected advancement by reaching $50,000$ rows in two additional episodes. Furthermore, the network is able to learn optimal polices on states projected to lower dimensions using the Feature Extraction network. Performance steadily decreases with dimensions as the network has less flexibility in learning the correct features from the original space but the computational demand was also reduce with smaller network sizes. However, surprisingly there was a decrease in performance when using the Eligibility Trace network. In fact performance increased as we approached a pure MM or TD update.

Our explanation for this result is that the network is already capable of function approximation through its distributed representation. So after each update, all state values are modified, however closer states should be affected more in the direction of the error and move closer to this return. This follows from the fact that states are

topographically related in the problem space and not generated independently as in our Gridworld problem. Therefore, updating a sequence of states can be seen as a form of eligibility trace where the parameter λ would be correlated with the density (distance between states) of valid states in all of state space. In summary our system outperform standard tabular Q-learning on the Gridworld problem and therefore supports our thesis that Associative memories can be used as a computational model to perform supervised, unsupervised and now reinforcement learning.

Chapter 5

Conclusion

In this work we showed that previous models of associative memories were capable of learning both supervised and unsupervised learning task with properties that makes these models ideal for a computation model of human learning. These include dynamic properties such as establishing fixed-points, strange attractors, Hebbian rule, pattern completion, noise correction, self organizing with competitive learning. Our objective was then to incorporate an additional paradigm reinforcement learning into the model while preserving its current abilities. We proposed a new model to solve reinforcement learning problems by an associative memory network. We achieved this by the novel idea of representing state returns as the negation of the network state's energy level. However this required a modification of the learning rule to correct for the new error. A modified Hebbian inspired learning rule that generalizes previous rules and as a consequence preserve the network properties. The effect this had on the energy function is to scale existing fix points to higher boundaries. Using this network we were able to outperform a native reinforcement algorithm, Q-learning, on a stochastic Gridworld task with p-value of $4.5 \cdot 10^{-24}$ and completed up to $50,000$ rows in a modified Tetris game.

Three additional networks were suggested that includes a new policy based on dynamical network principles using chaos. However, a suitable tradeoff between exploration and exploitation could not be found to get competitive results with the ϵ-greedy action selection. The other was extracting features of the original problem state dimension to a lower dimension state for learning. This proved ideal as a trade off between the complexity of

the state space and the order of computations needed to learn correct actions.

Our contribution to the research field is network V and representing states returns as the negation of the network's energy function. We were able to encode Q-learning in the network and perform reinforcement learning. In addition, the dynamic policy of encoding action selection as a searching process, similar to that of the *max* operator used in ϵ-greedy polices. Ideally this policy should be used since it is constructed from features native to AM. Also, the technique of combining feature extraction into the RL algorithm instead of a separate preprocessing step that is done manually.

5.1 Future Work

Possible future directions we could go in includes determining a suitable tradeoff between exploration and exploitation in the dynamic policy as mentioned before. This would be an ideal policy to use since we will be able to remove stochastic elements from the network and rely solely on chaos for any diversity. The advantage of this from a biological perspective is it would be make more sense if we are to believe the brain is capable of being modeled by a deterministic machine.

We can also expect to extend the representation of states and actions from the discrete to include their continuous forms since the underlying AM network supports continuous states. Although discrete problems come up often in contrive examples the real world is full of continuous problems worth exploring. One common method is to discretize continuous space to discrete intervals but this has a problem of scaling due to the large number of intervals required to guarantee good coverage of the original space or knowing in advance the optimal interval size. A solution based off the continuous form of the Bellman [52, 56] equations might be ideal in this case.

Furthermore, we notice the network uses a union of the parameters from associative memory and reinforcement learning. It would be beneficial to reduce the number of parameters by combining them or derived others from hyper parameters. We have intentionally decided to focus on direct methods despite the advantage of having a model of the environment. We will consider this a possible area for future work as a basis for planning

problems. In that case we will build a model with a state value function instead of action values. The difficulty would be in estimating probabilistic outcomes from experiences. In addition to these, there are currently many open problems in RL that are under active research that we have not considered but techniques which we could incorporate into our network at a future time such as hierarchical actions [57] and learning a model [43].

Bibliography

[1] K. Kaneko and I. Tsuda. Chaotic itinerancy. *Focus Issue*, 13(3):926–936, 2003.

[2] S. Chartier and M. Boukadoum. A sequential dynamic heteroassociative memory for multistep pattern recognition and one-to-many association. *IEEE Trans. Neural Networks*, pages 59–68, 2006.

[3] A. Sharda and J. Freeman. How brains make chaos in order to make sense of the world. *Behavioral and Brain Sciences*, 10:161–195, 1987.

[4] J. Freeman. The creation of perceptual meanings in cortex through chaotic itinerancy and sequential state transitions induced by sensory stimuli. *Ambiguity in mind and nature*, 64:421, 1995.

[5] C. Watkins and P. Dayan. Learning from delayed rewards. *PhD Thesis*, 1989.

[6] R. Sutton. Temporal credit assignment in reinforcement learning. *PhD Thesis*, 1984.

[7] R. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[8] S. Singh and R. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22(1-3):123–158, 1996.

[9] D. Hebb. *The organization of behavior.* New York: Wiley, 1949.

[10] Kosko B. Bidirectional associative memories. *IEEE Trans. Systems, Man and Cybernetics*, 18(1):49–60, 1988.

[11] S. Chartier, P. Renaud, and M. Boukadoum. A nonlinear dynamic artificial neural network model of memory. *New Ideas in Psychology*, 26(2):252–277, 2008.

[12] S. Zhu and D. Hammerstrom. Reinforcement learning in associative memory. In *Proceedings of the International Joint Conference on Neural Networks*, volume 2, 2003.

[13] J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the United States of America*, 79(8):2554–2558, 1982.

[14] J. Hopfield. Neurons with graded response have collective computational properties like those of two-state neurons. *Proceedings of the National Academy of Sciences of the United States of America*, 81(10):3088–3092, 1984.

[15] Rojas R. *Neural Networks: A Systematic Introduction.* Springer, 1996.

[16] C. Leung. Optimum learning for bidirectional associative memory in the sense of capacity. *IEEE Trans. System, Manual, Cybernetics*, 24(5):791–795, 1994.

[17] D. Shen and Jr J. Cruz. Encoding strategy for maximum noise tolerance bidirectional associative memory. *IEEE Trans. Neural Networks*, 16:293–300, 2005.

[18] J. Hopfield and D. Tank. 'neural' computation of decisions in optimization problems. *Biological Cybernetics*, 52(3):141–152, 1985.

[19] D. Ackley, G. Hinton, and T. Sejnowski. A learning algorithm for boltzmann machines. *Cognitive Science*, 9:147–169, 1985.

[20] S. Grossberg. Some nonlinear networks capable of learning a spatial pattern of arbitrary complexity. *Proc. Nat. Acad. Sci.*, 60:368–372, 1968.

[21] R. McEliece, E. Posner, E. Rodemich, and S. Venkatesh. The capacity of the hopfield associative memory. *Information Theory, IEEE Trans.*, 33(4):461– 482, 1987.

[22] L. Personnaz, I. Guyon, and G. Dreyfus. Information storage and retrieval in spin-glass like neural networks. *J. Physique Letter*, 46:359–365, 1985.

[23] S. Chartier, G. Giguere, P. Renaud, J. Lina, and R. Proulx. Febam: a feature-extracting bidirectional associative memory. In *Proceedings of International Joint Conference on Neural Networks*, pages 97–102, Orlando, Florida, USA, 2007.

[24] G. Giguere, S. Chartier, R. Proulx, and J. Lina. Category development and reorganization using a bidirectional associative memory-inspired architecture. In *Proceedings of ICCM - Eight International Conference on Cognitive Modeling*, pages 97–102, Oxford, UK, 2007.

[25] S. Chartier and G. Giguere. Autonomous perceptual feature extraction in a topology-constrained architecture. 2008.

[26] S. Chartier, M. Boukadoum, and M. Amiri. Bam learning of nonlinearly separable tasks by using an asymmetrical output function and reinforcement learning. *IEEE Trans. Neural Network*, page 13, 2008.

[27] S. Haykin. *Neural Networks: A Comprehensive Foundation.* New York: Wiley, 1999.

[28] T. Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43:59–69, 1982.

[29] S. Chartier and Proulx R. Ndram: nonlinear dynamic recurrent associative memory for learning bipolar and nonbipolar correlated patterns. *IEEE Trans. Neural Networks*, 16(6):1393–1400, 2005.

[30] Vidyasagar M. Discrete optimization using analog neural networks with discontinuous dynamics. In *Proc. of International Conference Automation, Robotics, Computer Vision*, Singapore, 1994.

[31] S. Chartier and M. Boukadoum. a bidirectional heteroassociative memory for binary and grey-level patterns. *IEEE Trans. neural networks*, 17(2):385–396, 2006.

[32] Z. Xu, Y. Leung, and X. He. Asymmetric bidirectional associative memories. *IEEE Trans. Systems, Man and Cybernetics*, 24(10):1558–1564, 1994.

[33] I. Tsuda. Toward an interpretation of dynamic neural activity in terms of chaotic dynamical systems. *Behavioral and Brain Sciences*, 24:793–847, 2001.

[34] J.. Amit. Modeling brain function: The world of attractor neural networks. 1989.

[35] T. Wills, C. Lever, F. Cacucci, N. Burgess, and J. O'Keefe. Attractor dynamics in the hippocampal representation of the local environment. *Science*, 308(5723):873–876, 2005.

[36] Glenn Elert. *The Chaos Hypertextbook.* 1995.

[37] M. Adachi and K. Aihara. Associatie dynamics in a chaotic neural network. *Neural Networks*, 10(1):83–98, 1997.

[38] A. Araujo, L. Bueno, and M. Compos. Dynamic behaviors in chaotic bidirectional associative memory. *Journal of Intelligent & Fuzzy Systems*, 18:513–523, 2007.

[39] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, 2nd Edition.* Prentice Hall, 2003.

[40] R. Sutton and Barto A. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning).* The MIT Press, 1998.

[41] R. Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, 6, 1957.

[42] R. Howard. Dynamic programming and markov processes. *The M.I.T. Press*, 1960.

[43] L. Kaelbling and M. Littman. Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[44] C. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.

[45] J. Peng and R. Williams. Incremental multi-step q-learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 226–232, San Francisco, 1994. Morgan Kaufmann.

[46] A. McGovern and R. Sutton. Towards a better q(lambda). *Presented at the Reinforcement learning Workshop*, 1997.

[47] R. Brafman and M. Tennenholtz. r-max - a gernal polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3:213–231, 2002.

[48] J. Gittins and D. Jones. A dynamic allocation index for the sequential design of experiments. *Progress in Statistics*, 1974.

[49] G. Palm. On associative memory. *Biological Cybernetics*, pages 19–31, 1980.

[50] N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *Automaic Control, IEEE Trans.*, 42(5):674–690, 1997.

[51] P. Stone and R. Sutton. Reinforcement learning for robocup soccer keepaway. *Adapative Behavior*, 13(3):165–188, 2005.

[52] D. Bertsekas and J. Tsitsiklis. *Neuro-dynamic programming.* MIT, 1996.

[53] C. Fahey. Tetris. *http://www.colinfahey.com*, 2003.

[54] E. Demaine, S. Hohenberger, and D. Liben-Nowell. Tetris is hard, even to approximate. *Technical Report MIT-LCS-TR-865*, 2002.

[55] H. Burgiel. How to lose at tetris. *Mathematical Gazette*, 1997.

[56] K. Doya. Reinforcement learning in continuous time and space. *neural computation*, 12:219–245, 2000.

[57] A. Barto and S. Mahadevan. Recent advances in hierarchial reinforcement learning. *Discrete Event Dynamic Systems*, 13(4):1573–7594, 2003.

Appendix A

Source Code

Standard Functions

```
sgn := Function[x, Map[If[# >= 0, 1, -1] &, x]];
bound := Function[u,
   Map[Function[x, If[x > 1, 1, If[x < -1, -1, x]]], u]];
auto := Function[{m, u, q, \[Eta]},
   With[{U = next[m, u], Q = energy[m, u]},
    With[{err = Outer[Times, u, u] - Outer[Times, U, U]},
     m + \[Eta] ((-q) - (-Q)) err]]];
AUTO := Function[{m, u, \[Eta]},
   With[{U = next[m, u]},
    With[{err = Outer[Times, u, u] - Outer[Times, U, U]},
     m + \[Eta]*err]]];
hetero :=
  Function[{m, u, U, v, V, \[Eta]},
   m + \[Eta]*Outer[Times, u - U, v + V]];
bamT := Function[{w, v, x, y, \[Eta]},
  With[{X = next[v, y], Y = next[w, x]},
   {hetero[w, y, Y, x, X, \[Eta]], hetero[v, x, X, y, Y, \[Eta]]}]]
bamR := Function[{w, v, x, y}, {next[v, y], next[w, x]}];
febamT := Function[{w, v, x0, k, \[Eta]},
   With[{y0 = wta[next[w, x0], k]},
    With[{x1 = next[v, y0]},
     With[{y1 = sgn[next[w, x1]]},
      {hetero[w, y0, y1, x0, x1, \[Eta]],
       hetero[v, x0, x1, y0, y1, \[Eta]]}]]]];
febamR[w_, v_, x_, n_, \[Delta]_: 0.01] :=
  Nest[next[w, next[v, #, \[Delta]], \[Delta]] &,
   next[w, x, \[Delta]], n];
wta[v_, k_: 1] :=
  With[{pos = Map[List, Sort[Take[Ordering[v], -k]]]},
   MapAt[1 &, Map[-1 &, v], pos]];
(* \[Delta] = 1.65 unrestrained chaos; \[Delta] = 1.45 restrained \
chaos; \[Delta] = 0.1 behaved *)

next[m_, v_, \[Delta]_: 0.01] :=
```

```
  With[{mv = Map[bound, m.v]}, (\[Delta] + 1) mv - \[Delta]*mv^3];
eg := -energy[#1, #2] &;
energy[m_, x_, \[Delta]_: 0.01] :=
  With[{y = next[m, x]}, (* lowest energy is -Max[n,
   m] *)
   -(x.y - (1 + \[Delta])*x.m.y + (\[Delta]/2)*x.(m.y)^3 +
       0)/2];
```

Network V

```
AMLearning := Function[{n, dimS, dimA, \[Lambda]},
   Module[{s, S, a, A, q, Q, r, ia, iA,
     v, sa, SA, \[Eta] = 10^{-4},
     step = 1, ep = 1, R = 0, dim = dimS + dimA, policyz, Ar = {}},
    v = ConstantArray[0, {dim, dim}];
    Print["Ep\t", Dynamic[ep], "\tStep\t", Dynamic[step], "\tRe/Ep\t",
      Dynamic[R/(ep - 1) // N]];

    policy := Function[{m, s, \[CurlyEpsilon]},
      If[RandomReal[] > \[CurlyEpsilon],
       With[{act = Map[energy[m, Join[s, #]] &, actions]},
        First[RandomChoice[Position[act, Max[act]]]]],
       RandomInteger[{1, numA}]]];

    For[step = 0; ep = 1, ep <= n, ep++,
     s = start[];
     While[Not[terminal[]],
      ia = policy[v, s, 1/ep];
      sa = Join[s, actions[[ia]]];
      {r, S} = env[ia];

      iA = policy[v, S, 0];
      SA = Join[S, actions[[iA]]];
      q = energy[v, sa];
      Q = If[terminal[], 0, energy[v, SA]];
      With[{\[CapitalDelta] = r + \[Gamma]*Q - q},
       v =
        Nest[auto[#, sa, q + \[Alpha]*\[CapitalDelta], \[Eta]] &, v,
         1]];
      s = S; R = R + r; step = step + 1;
      ];
     Ar = Append[Ar, R/ep];
     step = 0;
     ];
    Ar]];
```

Network VPTF

```
AMLearning := Function[{n, dimS, dimA, \[Lambda]},
   Module[{s, S, a, A, q, Q, r, ia, iA,
     v, c, C, e, E, f, F,
     sa, SA, \[Eta]2 = 0.001, \[Eta] = 0.00001*0.01,
     step = 1, ep = 1, \[Delta], R = 0, dim = dimS + dimA, statea,
     Ar = {}, As = {}},

    (*Print["Ep\t",Dynamic[ep],"\tStep\t",Dynamic[step],"\tRe/Ep\t",
    Dynamic[R/(ep-1)//N]];*)

    Print[Dynamic[ep], "\t", Dynamic[step], "\t",
     Dynamic[R/(ep - 1) // N], "\t", Dynamic[R]];
    {f, F} = {RandomReal[{-0.1, 0.1}, {dimS, dimState}],
      RandomReal[{-0.1, 0.1}, {dimState, dimS}]};
    e = v = c = C = ConstantArray[0, {dim, dim}];

    policy := Function[{m, s, \[CurlyEpsilon]},
      If[RandomReal[] > \[CurlyEpsilon],
       With[{act = Map[energy[m, Join[s, #]] &, actions]},
        First[RandomChoice[Position[act, Max[act]]]]],
       RandomInteger[{1, numA}]]];

    statea := Function[s,
      If[dimState == dimS, s,
       {f, F} =
        Nest[febamT[#[[1]], #[[2]], s, Round[dimS/3], \[Eta]2] &, {f,
          F}, 1];
       sgn[febamR[f, F, s, 1]]]];

    For[step = 0; ep = 1, ep <= n, ep++,
     s = statea[start[]];
     While[Not[terminal[]],
      ia = policy[v, s, If[ep >= 90, 0, 1/ep]];
      sa = Join[s, actions[[ia]]];

      If[step == 0, {c, C} = bamT[c, C, sa, sa, \[Eta]2]];
      {r, S} = env[ia]; S = statea[S];

      iA = policy[v, S, 0]; SA = Join[S, actions[[iA]]];
      q = energy[v, sa];
      Q = If[terminal[], 0, energy[v, SA]];
      \[Delta] = r + \[Gamma]*Q/ep - q;

      {c, C} = bamT[c, C, SA, sa, \[Eta]2];
      e = auto[e, sa, 1, \[Eta]];
      If[\[Lambda] <= 0,
```

```
   v = Nest[auto[#, sa, q + \[Alpha]*\[Delta], \[Eta]] &, v, 1],
   Module[{x = sa, collect = {}},
    While[Not[MemberQ[collect, x]] && Length[collect] < dimS,
     q = energy[v, x];
     E = energy[e, x];
     v = auto[v, x, q + \[Alpha]*\[Delta]*E, \[Eta]];

     If[x[[1 ;; dimS]] == s && x != sa,
      e = auto[e, x, 0, \[Eta]],
      e = auto[e, x, \[Gamma]*\[Lambda]*E, \[Eta]]];

     collect = Append[collect, x];
     x = sgn[next[c, x]]]
    ]];

  s = S; R = R + r; step = step + 1;
  ];
 Ar = Append[Ar, R];
 step = 0;
 ];
Ar]];
```