Temporal Difference Learning with Interpolated Table Value Functions

Simon M. Lucas, *Senior Member, IEEE*

***Abstract*— This paper introduces a novel function approximation architecture especially well suited to temporal difference learning. The architecture is based on using sets of interpolated table look-up functions. These offer rapid and stable learning, and are efficient when the number of inputs is small. An empirical investigation is conducted to test their performance on a supervised learning task, and on the mountain car problem, a standard reinforcement learning benchmark. In each case, the interpolated table functions offer competitive performance.**

I. Introduction

One of the key problems in game strategy learning is how an agent can best learn in a largely unsupervised manner via interactions with its environment. The payoff for a particular action may be far removed in time and in state-space from the action's execution.

There are many ways to formulate solutions to reinforcement learning problems, but one of the most general and easily applied is the value function method. At each time the agent considers a set of possible actions, and using a model of the environment, maps the execution of each action to a possible future state, creating a set of future states S. The value function is then applied to calculate the value $v(s)$ of each state $s \in S$, and the action that leads to the highest value is selected.

A key problem for reinforcement learning is to choose the most appropriate function approximation architecture. Making a suitable choice can mean the difference between learning very effectively, and hardly learning at all. Table-based functions are known to work well with Temporal Difference Learning, but can only be applied directly when the number of inputs is relatively small, and take on a limited number of values (or can be discretised to a limited number of values without preventing good performance).

N-tuple sampling can be used in many cases to address problems that involve a large number of inputs. N-Tuple systems operate by randomly sub-sampling the input space, taking n inputs for each sub-network. Each sub-network is simply an independent look-up table. For a particular input vector, the overall system output is some combination of the values indexed by each individual table.

While n-tuple networks [1], [2] date back nearly 50 years, they have only recently been applied to game strategy learning [3], where they were shown to be very effective as the function approximation architecture when learning to play Othello using temporal difference learning. These n-tuple networks have very significantly outperformed weighted piece counters, standard multi-layer perceptrons, and spatially arranged multi-layer perceptrons of the type used in Blondie [4] on the Othello neural network server[1], used for the IEEE WCCI 2008 Othello Competition.

The original motivation for the current paper was to apply similar techniques to problems where the input space is continuous rather than discrete. A standard way to do this for n-tuple networks is to use the so-called CMAC + Grey coding [5], [6]. However, this proved to be ineffective on the problems tested in this paper. Instead, a new type of function approximator has been developed, the bi-linear interpolated table function, which is the main contribution of the paper.

The rest of this paper is structured as follows. Section II briefly discusses some commonly used function approximators, and introduces the interpolated table function. Section III presents results on a supervised regression problem, the continuous intertwined spirals. Section IV presents results on a standard reinforcement learning problem, the Mountain Car. Section V concludes.

II. Function approximation architectures

There are two main classes of function approximator: global and local. Global approximators such as multi-layer perceptrons divide the input space into regions using hyperplanes. Each layer consists of a set of units, where each unit computes the scalar product of its weight vector and it's input vector before passing it through a non-linear activation function. The weight vector of each unit determines the position, orientation and slope of the non-linear activation function. Multiple layers allow any continuously measurable function to be approximated. The approach is global in the sense that small changes to a single weight can affect the entire functional map. This means that the knowledge embodied in the network can be disrupted as new information is learned. This problem is especially serious for on-line learning problems. However, this global nature also means that MLPs do not suffer from the so-called curse of dimensionality.

Local approximators such as radial basis functions (RBFs), and nearest neighbour classifiers work by using a distance measure to calculate the proximity of the input pattern to

The author is with the School of Computer Science and Electronic Engineering, University of Essex, Colchester, United Kingdom. (phone: +44-1206-872048; fax: +44-1206-872788; email: sml@essex.ac.uk).

[1] http://algoval.essex.ac.uk:8080/othello/html/Othello.html

each RBF centre or each pattern sample. Methods based on distance measures suffer from the curse of dimensionality, in that the number of network nodes needed to accurately map the space can grow exponentially with respect to the number of inputs. Dimensionality reduction methods such as principal component analysis can be used to alleviate the effects of this, but can also introduce problems of their own. Support vector machines can be either local or global, depending on the kernel used.

For traditional supervised learning tasks there is usually a fixed training set to learn, and algorithms such as back propagation can be used to learn these datasets effectively, but often require many iterations through the dataset in order to do so.

This paper is concerned with architectures that are most suitable for learning to play games while playing them i.e. on-line learning. It is also possible (and often very effective) to learn game strategy from databases of previously played games, in which case it becomes a standard supervised learning problem. However, the on-line learning method has certain advantages, such as allowing the behaviour to be adapted to a particular opponent. It can also be used for novel games where no expertise or database of games exists, and is especially interesting from a machine-learning perspective.

A. Table Functions

When they can be applied, table functions work extremely well. The input values are used to directly index a table location, and the value at that index is updated directly on the basis of the current learning signal. For small discrete input spaces this is simple and direct to apply. For large discrete, or continuous input spaces the number of table entries becomes too large to apply this technique directly, and some type of approximation must be used.

This paper directly tackles the problem of continuous spaces with a small number of inputs, and briefly discusses how the results may be generalised to continuous spaces with a large number of inputs.

The simplest way to use a table with continuous inputs is to take a discretised version of each input, and use the discretised values to index a multi-dimensional table. If we have a two-dimensional function $f(x, y)$ and a discretisation function q', then we use a table $t[q'(x)][q'(y)]$ to represent the function. Clearly, for a given level of discretisation, the table size grows exponentially with respect to the number of inputs.

Using this method as a value function has a significant limitation, however: there is a tension between loss of gradient, and poor generalisation. If the discretisation is too coarse, then many possible future states will have exactly the same value, leaving no principled way to choose between apparently equal alternatives. This situation is illustrated in Figure 1. The possible next states from $s(t)$ are $s1, s2, s3$, but the value function will always be equal for $s1$ and $s2$ so there is no principled way to choose between them. If the discretisation is too fine, then the system may learn slowly, as it will take longer to explore (and hence learn a value for)

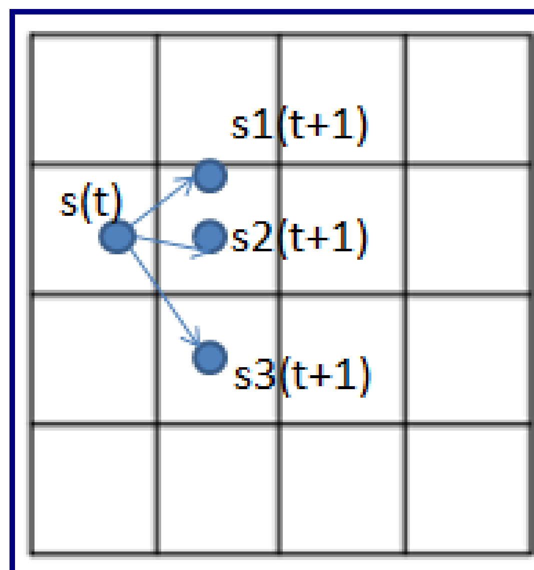

Fig. 1. A coarse grid that is unable to store distinct values for states $s1$ and $s2$.

all the feasible states. Over-fine discretisation also leads to excessive memory requirements.

The conventional way around these problems of directly using tables in reinforcement learning is to use CMAC (cerebellar model articulation controller) tables. This is identical to the discretisation approach described above, except that many (e.g. 10) randomly offset discretisations are now used, with each input of each one subject to a different random offset. This method is described in Sutton and Barto [7], and illustrated in Figure 2 for convenience. The left hand image shows a standard discretisation of a two-dimensional space, where each point is allocated to the table square that it falls in. The right hand image shows a CMAC table with five randomly overlapping discretisations. When implementing this, each discretisation can index the same table of values (shared), or a different one (separate), though for the experiments in this paper the shared version always performed better than the separated one. CMAC tables still have flat areas with no gradient, but the flat areas are now much smaller. In general, if the original flat regions were hypercubes of side length l, they become hypercubes with average side length l/c assuming that we have c CMAC discretisations. The main downside of the CMAC table is that the discretisation and lookup process must be repeated for each discretisation, and that there is some random dispersion between neighbouring points in the input space.

B. Interpolated Table Lookup

There are many possible ways to perform interpolated table lookup, but the approach we take here is to implement the d-dimensional generalisation of bi-linear interpolation. Despite the name, the interpolation surface is not linear except for the one-dimensional case of $d = 1$, but is in general a polynomial of degree d.

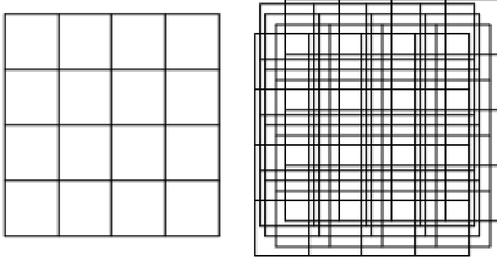

Fig. 2. Standard table discretisation (left) versus CMAC discretisation (right).

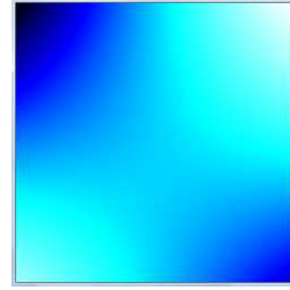

Fig. 3. The bi-linear interpolated output between four table entries (the four corners) with values of $-1.0, 1.0, 0.8, 0.2$.

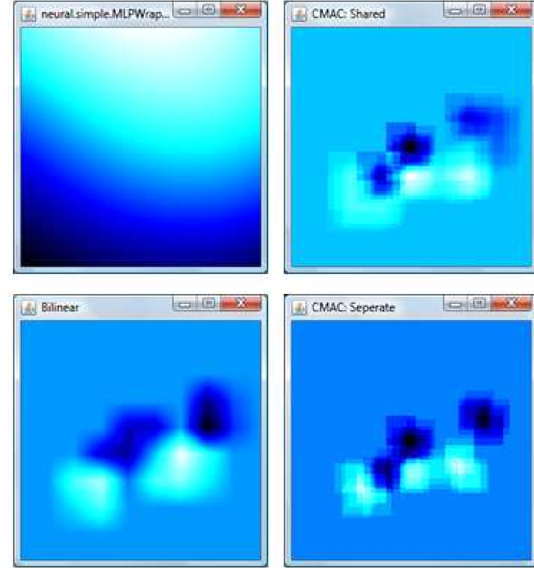

Fig. 4. In order from left to right, top to bottom, an MLP, a shared CMAC table, a bi-linear interpolated table, and a separated CMAC table, each trained with the same six patterns.

Given a d-dimensional input vector $\vec{v}$, each dimension of the vector is first quantised into upper and lower integers, each of n_q possible values.

The functions used to perform quantisation are as follows (this simple form assumes that it is done in the same way for each dimension, but it is straightforward to use a range and quantisation that is specific to each level). The first step is to map the input value a into the range $[0, (n_q - 1)]$. This is done with the following expression to find the value $q(a)$:

$$q(a) = n_q(a - \text{min})/(\text{max} - \text{min})$$

From the real value $q(a)$ we then find the integers directly below (q_l) and above (q_u) using the floor and ceiling functions:

$$\begin{aligned} q_l(a) &= \lfloor q(a) \rfloor \\ q_u(a) &= \lceil q(a) \rceil \end{aligned}$$

The residue $r(a)$ is the fractional part left over after subtracting off the lower integer:

$$r(a) = q(a) - q_l(a)$$

For the two dimensional case the bi-linear interpolated calculation $f_t(x, y)$ is as follows for point x, y, where x and y are real numbers in the range *min* to *max*, as used above. The value at each table entry indexed by i, j is given by $t[i][j]$ and used as follows:

$$\begin{aligned} f_t(x, y) &= (1 - r(x))(1 - r(y))t[q_l(x)][q_l(y)] \\ &+ r(x)(1 - r(y))t[q_u(x)][q_l(y)] \\ &+ (1 - r(x))r(y)t[q_l(x)][q_u(y)] \\ &+ r(x)r(y)t[q_u(x)][q_u(y)] \end{aligned}$$

The generalisation to a d dimensional space is straightforward, leading to a d-degree polynomial with 2^d terms.

Figure 3 shows the bi-linear interpolated values between four corners of a grid square with values of $-1.0, 1.0, 0.8, 0.2$ respectively. By comparison, a standard table would have the same value over the entire area of each grid cell.

Figure 4 gives some insight into the nature of the surfaces represented by an MLP, a CMAC table, and a bi-linear interpolated table. Each approximator was trained with a single pass through a dataset with six patterns, three positive, and three negative, with the data in the format (x, y, target) shown in Table I. In the case of the CMAC architecture, 10 overlapping grids were used. In each of the table functions (CMAC and bi-linear), a 10×10 grid was used. The evidence of the training patterns can be clearly seen for all the table-based methods, but not for the MLP, which would require many iterations to learn this. For many game-based learning problems, the one-shot learning capability of the table-based methods is an important consideration.

III. Test Functions

To evaluate the performance of the interpolated table, and to gain more insight into the nature of the learned function approximations, the system was first tested on a 2-dimensional function to enable easy visualisation of the results. The architectures tested were MLPs, N-Tuple networks (CMAC+Grey input coding), standard table lookup,

TABLE I

SMALL TEST DATASET TO ILLUSTRATE WHAT EACH ARCHITECTURE LEARNS AFTER A SINGLE PASS THROUGH THE DATA.

x	y	f
0.5	0.5	-1.0
0.7	0.6	1.0
0.4	0.6	-1.0
0.3	0.7	1.0
0.8	0.4	-1.0
0.5	0.6	1.0

Architecture	MSE
MLP	0.13
N-Tuple (CMAC + Grey)	0.30
Standard Table	0.08
CMAC (Shared)	0.01
Bi-Linear	0.006

TABLE II

MEAN SQUARE ERROR (MSE) FOR EACH ARCHITECTURE AFTER 50,000 RANDOMLY CHOSEN TRAINING PATTERN PRESENTATIONS.

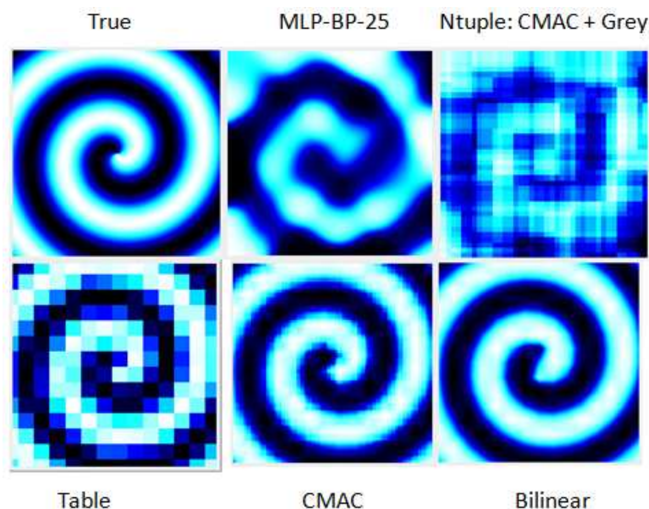

Fig. 5. Learning a continuous spiral function. From left to right, top to bottom. (a) The spiral function to be learned; (b) an MLP after 50,000 training iterations of backprop; (c) a CMAC encoded N-Tuple network; (d) a 15×15 table, (e) CMAC table (10 of 15×15), and (f) a 15×15 bi-linear interpolated table.

CMAC tables, and bi-linear interpolated tables. Each architecture was trained using error back-propagation. All the table methods used a 15×15 grid.

The back-propagation training regime was as follows. At each iteration a single input vector was presented to the network, the network was run in forward mode, and the output compared with the output of the function to be approximated. The squared error was added to a running total, and the error derivative was back-propagated through the network. Every 1,000 iterations, the average error was reported and reset. This regime was chosen in order to give a robust summary of progress, and to create similar conditions to the way these function approximators are used within temporal difference learning algorithms, where the same input and target vectors might never be repeated, unlike the repeated iterations through a fixed dataset used in standard supervised learning.

The chosen test function is a continuous spiral. This is defined as follows in polar coordinates (r,θ), where ω is a constant proportional to the number of turns:

$$f(x,y) = \sin(\theta + r\pi\omega)$$

The complexity of this function can be varied smoothly by varying ω. Spirals with more turns in the given input range are more intricate, and more difficult to learn. For these experiments ω was set to 6.0, and the function was plotted with x and y in cartesian coordinates in the range $[-0.5, 05]$.

Figure 5 shows a visualisation of the performance of each approximator when trained with back-propagation.

Table II shows the mean squared error averaged over the last 1,000 of the 50,000 training presentations for each architecture. The CMAC and bi-linear interpolated tables perform best on this task. They have the smallest mean square error, and they also fit the best visually.

These table-based approximators are easy to use. The resolution of the table can be chosen directly, and the effects of that choice are easy to analyse. Furthermore, there are some simple tricks that can be applied to boost learning, such as Gaussian smoothing.

IV. RESULTS ON THE MOUNTAIN CAR PROBLEM

The mountain car task was introduced by Moore [8], and is also described in ([7], p. 214). The problem is illustrated in Figure 6. The task is to drive a car to reach a goal at the top of a hill. The state of the car is defined by two variables: it's position (s) and it's velocity (v). The goal is shown as the rightmost vertical line at ($s = 0.5$). There is a barrier to the left (the leftmost vertical line) at ($s = -1.2$) which the car stops dead at. At each time step, there are three possible actions: accelerate left ($a_t = -1$), accelerate right ($a_t = +1$), or apply no acceleration ($a_t = 0$). The car engine has insufficient force to directly overcome gravity, so the problem is a bit deceptive: in order to reach the goal as quickly as possible, it may be necessary to accelerate (apply a force) away from the goal in some circumstances. The following are the standard mountain-car equations and specify the update rules for position (s) and velocity (v) where the *cos* function is in radians:

$$\begin{aligned} s_{t+1} &= \text{bound}_s(s_t + v_{t+1}) \\ v_{t+1} &= \text{bound}_v(v_t + 0.001a_t - 0.0025\cos(3s_t)) \end{aligned}$$

The bounds for s and v are as follows:

$$\begin{aligned} -1.2 &\leq s_{t+1} \leq 0.5 \\ -0.07 &\leq v_{t+1} \leq 0.07 \end{aligned}$$

Note that the equations treat the problem as having a single dimension even though it is sketched out in two dimensions. The equations deal only with the horizontal components of

the position, velocity, and acceleration and should be seen as sufficient to provide a simple benchmark problem rather than an accurate simulation of the scenario.

Each epoch (trial) starts with the car having a random position and velocity drawn uniformly within the above bounds. Each trial terminates when the car reaches the goal, or after 2500 steps. The best solutions for this particular configuration of the problem solve the task in approximately 52 steps on average (each step has a cost of 1, so this is sometimes referred to as a score of -52).

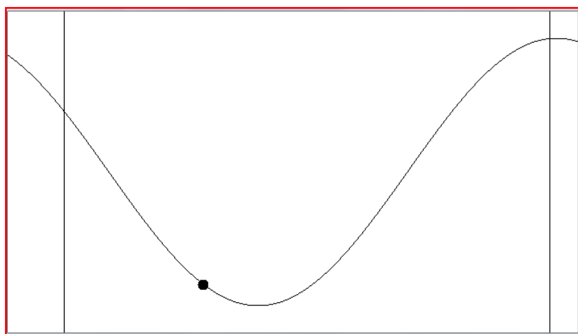

Fig. 6. The Mountain car problem. The task is to reach the line at the hill top on the right, but the engine force is weaker than gravity.

The mountain car problem is considered to be reasonably hard for neural networks to solve, and was used as a benchmark by Whiteson and Stone [9] as an initial test for their NEAT+Q evolutionary / temporal difference hybrid learning algorithm. NEAT is an abbreviation for Neuro-Evolution with Augmenting Topologies. The $+Q$ denotes the addition of Q-learning, a particular type of temporal difference learning. More recently the author has shown [10] that the mountain car task (in its standard form as used in this paper and in most other papers) actually has a trivial solution which is close to optimal. The trivial solution is to always choose the action leading to the state with the highest speed (irrespective of whether the direction is left or right). Nonetheless, the task is suitable for our current purpose of evaluating the performance of alternative function approximators within the same training regime.

The table-based architectures described in section II were applied to this problem, and each was trained for 2,000 epochs (the CMAC tables may require 10,000 epochs to reach optimal performance) in order to test their ability to learn relatively quickly. The method used was TD(0) [7], with the learning rate (α) set to 0.1 and the exploration parameter (ϵ) also set to 0.1. A standard reward scheme was used with a reward of 0 at the goal state and -1 everywhere else. This scheme helps encourage exploration during the learning process, since states that are frequently visited become increasingly negative unless they are close to the goal.

The results are shown in table III, with standard errors in parentheses. The best performing methods are bi-linear and CMAC (shared), with the bi-linear method significantly outperforming all the other methods. The learned performance of 50.5 reached by the bi-linear interpolated table is similar (the difference is not statistically significant) to the Whiteson and Stone quoted averages of 52.75 (NEAT+Q) and 52.02 (CMAC), though NEAT+Q took hundreds of thousands of epochs to reach that performance. We verified that our CMAC implementation was able to achieve similar performance (approximately 52) after 10,000 epochs, but when restricted to 2000 epochs the interpolated table method was clearly the best method.

Architecture	mean (s.e.)
Standard Table	1008 (143)
CMAC (Separate)	81.8 (11.5)
CMAC (Shared)	60.0 (2.3)
Bi-Linear	50.5 (2.5)

TABLE III

MEAN PERFORMANCE (AND STANDARD ERROR) OVER 10 TRIALS FOR EACH ARCHITECTURE AFTER 2,000 LEARNING EPOCHS. EACH SYSTEMS USED A 15×15 GRID. THE CMAC SYSTEMS EACH USED 10 OVERLAPPING GRIDS.

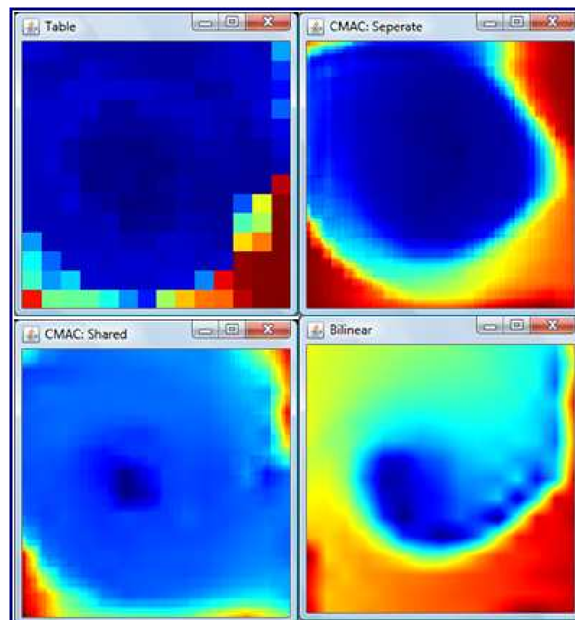

Fig. 7. Sample learned value functions for the mountain car problem after 2,000 epochs. In order from left to right, top to bottom: simple table, separate CMAC table, shared CMAC table, and interpolated table.

Figure 7 shows a sample learned solution for each of the types of function approximator. In each of the colour maps dark blue indiciates low values (strongly negative) while the lighter and red colours indicate relatively high values. The x-axis shows position (from left (min) to right (max)), and the y axis shows velocity (from top (max) to bottom (min)). The minimum velocity is when the speed is maximum but going left (away from the goal).

After 2,000 epochs these can look quite different in the detail and vary significantly on each run, but note how the bi-linear table is much smoother than the other types of table, while using the same grid size and hence the same amount

of memory as the simple table and the shared CMAC table.

V. Conclusions

When learning a value function for a control problem or a game, the function approximation architecture is a key consideration. This paper introduced a new architecture based on interpolated table lookup. The system was first tested on a supervised learning task where it was trained to learn a continuous spiral function. The interpolated tables were able to fit this better than the standard CMAC tables, both in terms of the mean square error, and in the observed smoothness of fit. On the widely used Mountain Car reinforcement learning benchmark, the interpolated table achieved high performance after relatively few episodes.

The interpolated table requires no extra storage, but does involve 2^d table lookups for a d-dimensional table, compared to just one lookup for a non-interpolated table, or c lookups for a CMAC table which has c discretisations of the input. Due to the fact that table size grows exponentially with d, d is often kept small for practical reasons.

The architecture can be applied to high-dimensional spaces by using an n-tuple style architecture, where the space is sampled by a set of low-dimensional interpolated tables, whose outputs are combined to form the overall output. Testing how well this works in practice is important future work.

In summary, interpolated tables are a simple modification of standard tables that enable smooth fitting of value functions. The results presented in this paper demonstrate competitive performance, and suggest that interpolated tables deserve further investigation.

References

[1] W. W. Bledsoe and I. Browning, "Pattern recognition and reading by machine," in *Proceedings of the Eastern Joint Computer Conference*, 1959, pp. 225–232.

[2] J. Ullman, "Experiments with the N-Tuple method of pattern recognition," *IEEE Transactions on Computers*, vol. 18, no. 12, pp. 1135–1137, December 1969.

[3] S. M. Lucas, "Learning to Play Othello with N-Tuple Systems," *Australian Journal of Intelligent Information Processing*, pp. 1 – 20, 2008.

[4] K. Chellapilla and D. Fogel, "Evolving an expert checkers playing program without using human expertise," *IEEE Transactions on Evolutionary Computation*, vol. 5, pp. 422 – 428, 2001.

[5] A. Kolcz and N. M. Allinson, "Application of the cmac input encoding in the N-tuple approximation network," *IEE Proceedings - Computers and Digital Techniques*, pp. 177 – 183, 1994.

[6] R. Rohwer and M. Morciniec, "A theoretical and experimental account of N-Tuple classifier performance," *Neural Computation*, vol. 8, pp. 629 – 642, (1996).

[7] R. Sutton and A. Barto, *Introduction to Reinforcement Learning*. MIT Press, 1998.

[8] A. Moore, "Efficient memory-based learning for robot control," Ph.D. dissertation, University of Cambridge, 1990.

[9] S. Whiteson and P. Stone, "Evolutionary function approximation for reinforcement learning," *Journal of Machine Learning Research*, vol. 7, pp. 877–917, 2006.

[10] S. M. Lucas, "Computational intelligence and games: Challenges and opportunities," *International Journal of Automation and Computing*, pp. 45 – 57, 2008.