UNIVERSITY OF CAMBRIDGE

Modular Neural Networks for Learning Context-Dependent Game Strategies

by

Justin A. Boyan

B.S., University of Chicago (1991)

**Submitted to the Department of Engineering and Computer Laboratory
in partial fulfillment of the requirements for the degree of**

**Master of Philosophy
Computer Speech and Language ProcessingSeptember 1992**

Signature of Author ....

Justin A. Boyan
17 August 1992

Modular Neural Networks for Learning Context-Dependent Game Strategies

by

Justin A. Boyan

Submitted to the Department of Engineering and Computer Laboratory
on 17 August 1992, in partial fulfillment of the
requirements for the degree of
Master of Philosophy
Computer Speech and Language Processing

Abstract

The method of temporal differences (TD) is a learning technique which specialises in predicting the likely outcome of a sequence over time. Examples of such sequences include speech frame vectors, whose outcome is a phoneme or word decision, and positions in a board game, whose outcome is a win/loss decision. Recent results by Tesauro in the domain of backgammon indicate that a neural network, trained by TD methods to evaluate positions generated by self-play, can reach an advanced level of backgammon skill.

For my summer thesis project, I first implemented the TD/neural network learning algorithms and confirmed Tesauro's results, using the domains of tic-tac-toe and backgammon. Then, motivated by Waibel's success with modular neural networks for phoneme recognition, I experimented with using two modular architectures (DDD and Meta-Pi) in place of the monolithic networks. I found that using the modular networks significantly enhanced the ability of the backgammon evaluator to change its strategic priorities depending on the current game context. My best modular backgammon network was entered in the 1992 Computer Games Olympiad in London, where it finished in second place.

Thesis Supervisor: Professor Frank Fallside
Title: Head of Information Engineering Division, Department of Engineering

Contents

List of Figures

Chapter 1

Background

Connectionism—the study of computational models inspired by models of the brain—has surged in popularity over the past decade, forging productive new links between the communities of Artificial Intelligence (AI), machine learning (ML), and statistical pattern recognition (SPR). This thesis lies at the intersection of these three areas, and I hope it will serve as a small contribution to all three.

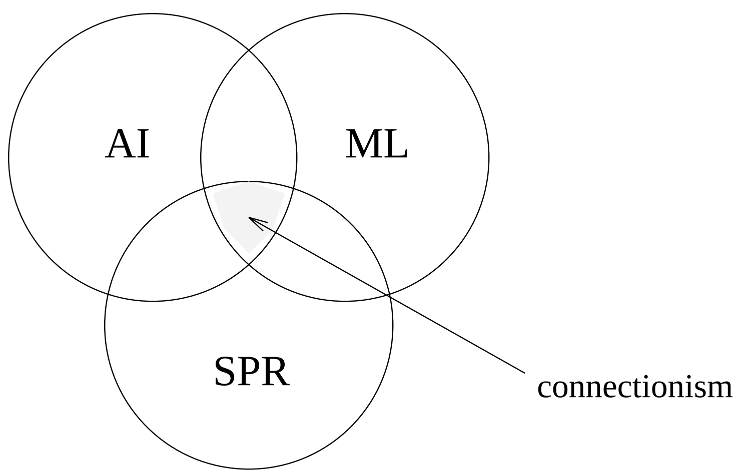

Figure 1-1: Connectionism in context

1.1 Speech Recognition and Connectionism

For both humans and machines, it seems likely that high-quality phoneme recognition must be the basis for successful speech understanding [BBB89, LHR89]. However, phoneme recognisers face three major difficulties:

- the acoustic evidence for a phoneme is a non-stationary signal, changing nonlinearly over time;

- a phoneme is context-dependent—it overlaps in time with surrounding phonemes, so there are rarely clear boundaries to indicate where one phoneme "ends" and another "begins"; and

- different realisations of the same phoneme can vary considerably along dozens of different dimensions.

Finding mathematical models and control architectures for overcoming these difficulties constitutes the frontier of speech recognition research.

1.1.1 Approaches to Phoneme Recognition

At present, the most successful framework for phoneme recognisers is the Hidden Markov Model [LHR89]. HMM's provide a quite flexible mechanism for modelling the uncertainties of phoneme recognition: a phone's time-varying properties are captured as a stochastic sequence of states in the model, and duration information is encoded in the self-loop transition probabilities for each state. Most importantly, both the state output probability distributions and the transition probabilities can be learned according to the iterative Baum-Welch training algorithm [Rab90]. The training procedure allows the model to make implicit generalisations to account for the variability of speech patterns; however, these generalisations are subject to the constraints of the model architecture. Despite the much higher degree of flexibility offered by HMM's than by earlier recognition models such as Discrete Time Warping [RL81], it is still the case that

> we do not know the true model for speech, but we are sure that it is not a piecewise stationary HMM, with some arbitrary choice of density function.[RMB+91]

Since the main objection to HMM's is that their representational flexibility is inadequate to model speech, researchers have sought more flexible models which are nevertheless trainable. One such model is the multilayer perceptron (MLP), which has enjoyed enormous popularity since the development of the error back-propagation training algorithm [RHW86]. The MLP is one of a class of "connectionist" or "artificial neural network" models—so-named because like human neurons, their computations are simple and localised; their actions may be executed in parallel; and the overall function computed by the network is distributed among the patterns of weights and activations throughout the network. Other architectures falling under the rubric of neural networks include Hopfield nets [Hop82], Kohonen nets [Koh88], LVQ networks [KBC88], and the Kanerva model [Kan88].

Biological analogies aside, the main attraction of the MLP model is that its parameters can be trained to define "an extremely flexible set of functions ... thus only weak assumptions are made about the input statistics" [RMB+91]. In a typical MLP, each unit in a "hidden layer" computes a weighted sum of the inputs, then applies a differentiable squashing function (usually

$\sigma(x) = 1/(1+e^{-x})$) to that sum. The result, the unit's "activation", is then "fed forward" to either another hidden layer or the output layer, where new unit activations are calculated similarly. The weights used in each weighted sum are the trainable parameters of the network; they are adjusted during training in an attempt to minimise an error function over weight space by the method of gradient descent. The error function is typically a sum-of-squares distance between the network's computed output and a target output, and the particular realisation of gradient descent for an MLP is the *error back-propagation algorithm* [RHW86].

The function computed by an MLP model is continuous with respect to its inputs; thus a model which performs well on the training input/output pairs will often be able to generalise to unseen inputs. However, there are many caveats to the MLP paradigm. For one thing, the size and shape of the MLP is not trainable; it is up to the researcher to choose a structure with enough free parameters (i.e. hidden units and weights) to model the problem, but not so many that the network overfits the training data and degrades its ability to generalise. For another thing, training a large net may require an inordinate amount of training data or computing resources. There are no theoretical guarantees of optimal convergence, since gradient descent may lead the network into only a local minimum of the error surface. Finally, even when a network does manage to successfully learn an input/output relation, the distributed nature of its solution reveals very little information about the problem's substructure. Nevertheless, neural nets have great appeal, and thousands of researchers in a wide variety of disciplines are currently using neural nets to solve real problems (see e.g. [Tou90]). Speech recognition is no exception: neural network-based approaches are quickly catching up to their more mature HMM-based counterparts.

1.1.2 Monolithic vs. Modular Neural Networks

The task of designing an appropriate MLP architecture for phoneme recognition is complicated by the dynamic nature of speech: the network must "represent temporal relationships between acoustic events, while at the same time providing for invariance under translation in time" [WSL89]. Typically, this is achieved by augmenting the MLP with either recurrent connections (back from the hidden layer to the input layer) [RF91], time-delay units (which provide input from multiple timesteps simultaneously) [WSL89], or some combination of the two [FLW90]. In training these architectures, the weight changes computed by error back-propagation must be averaged over time in order to enforce the constraint of shift-invariance. For Waibel's Time-Delay Neural Network (TDNN), this means that "the network is forced to discover useful acoustic-phonetic features in the input, regardless of when in time they actually occurred" [WSL89].

The basic TDNN [WSL89], trained to distinguish among the phonemes B, D, and G, made 4 times fewer classification errors than an HMM trained on the same data (98.5% correct versus 93.7%). These impressive figures result from a network topology which was carefully hand-tuned for this task and then trained with error back-propagation for approximately 100 hours on a supercomputer. However, Waibel and his colleagues also found that in order to scale up the

TDNN model to handle a large number of phonemes, simply increasing the breadth and depth of the network would not suffice. [WSS89] reports that a somewhat larger TDNN was able to achieve 98.3% correct recognition over a six-phoneme set (BDGPTK), but training required several *weeks* of supercomputer time. They concluded,

> we cannot hope for one single monolithic network to be trained within reasonable time as we increase task size and eventually aim for continuous, speaker-independent speech recognition. [WSS89]

Instead, a modular network architecture is necessary. As discussed in [JJB91], a modular structure composed of multiple neural networks provides several important advantages over a single monolithic network:

learning speed If a complex function decomposes naturally into several simpler functions in different regions of its domain, then a modular network should be able to learn relatively quickly both the set of simpler functions and a "switching function" to select among them. By contrast, learning in a monolithic network is slowed by *crosstalk*, the effect in which the units of the network receive conflicting error signals because the network is trying to learn multiple dissimilar functions at once.

generalisation Again, if the function to be learned is decomposable, then a modular architecture should generalise better than a monolithic architecture because its structure models the function more closely.

ease of development When the learning task is broken down into better-defined components, the networks may be designed and trained incrementally, allowing the developer to take full advantage of his or her domain knowledge.

Waibel's group applied techniques of modularisation to their original three-consonant, single-speaker TDNN in order to build all-consonant and multi-speaker recognition systems. They approached the all-consonant discrimination task as follows [WSS89]:

1. Apply expert domain knowledge to divide the Japanese consonants into small disjoint sets of similar phonemes (BDG, PTK, RWY, etc.), and train a separate TDNN on each set;

2. Train a somewhat larger "controller" TDNN to distinguish between the consonant subsets; and finally

3. Freeze the weights within each TDNN, link the structures together appropriately, and train only the inter-structure links with back-propagation.

The resulting conglomerate network achieved 95.0% recognition accuracy on the 18-consonant discrimination task, which rose to 95.9% after the frozen connections were freed and the network was trained to fine-tune all of its parameters. The excellent performance of the modular approach demonstrates the virtue of intelligently distributing the difficult classification decisions among subnetworks. A weakness of this particular scheme, however, is its heavy dependence on the controller subnetwork, whose errors dominated the error rate of the system. One approach to rectifying this imbalance would be to train a larger and more accurate controller net; however, the subnet may grow into just the sort of hard-to-train, monolithic network which modularity seeks to avoid.

A better approach to easing the dependence on the controller subnetwork would involve allowing the controller to make soft rather than hard classification decisions. The **Meta-Pi network** of Hampshire and Waibel [HW89] accomplishes this goal. A Meta-Pi network was trained to solve the scaled-up problem of *multi*-speaker, three-consonant recognition as follows:

1. Train a separate speaker-dependent TDNN on the task for each speaker (in this case six) and freeze these networks; and

2. Create and train the Meta-Pi "gating network", which is responsible for deciding *in what proportions* to mix the outputs of the speaker-dependent nets (hence its decisions are "soft").

An important point about Meta-Pi training is that the gating network is never provided with an explicit speaker identity; rather, its choice of speaker mixing proportions is determined solely with respect to the overall classification objective. This scheme is justified by Bayesian statistical theory, and also works very well in practice: [HW89] reported a 98.4% recognition rate on the six-speaker three-phoneme task, very nearly reaching the 98.7% average speaker-*de*pendent recognition rate. A drawback to the Meta-Pi formalism is that during recognition, all six of the subnetworks must be evaluated separately, so a sequential implementation runs 6 times more slowly than the "all-or-nothing" controller structure described earlier. Both of these modular designs take as a starting point the researcher's ability to specify the underlying task decomposition; methods for having the network learn the decomposition as well as the controller network are also currently being investigated [JJB91].

1.2 Game Playing

It has long been recognised that classical board games such as chess and checkers provide an excellent testbed for new theories in a variety of fields, including

- machine learning, e.g. Samuel's checkers program [Sam59];
- planning, e.g. Collins *et. al.*'s adaptive planning chess system [CBK89];
- pattern recognition, e.g. Lee and Mahajan's Othello program [LM88]; and

- neural networks, e.g. Tesauro and Sejnowski's backgammon program [TS89] and Robinson and Fallside's tic-tac-toe program [RF89].

Each of these games defines a domain which is easy to represent, program, and evaluate; yet expert-level play may require sophisticated abilities of planning, pattern recognition, and memory.

At the heart of most computer game algorithms is a *position evaluator* function. This function, given a board position as input, ideally returns the expected payoff from that board position given future optimal play by both sides. To decide on its move in a given situation, the game-playing program generates a list of all possible board positions resulting from the current legal moves; evaluates each with the static evaluator; and selects the move which results in the board with the highest evaluation. If the position evaluator function for a game is known accurately, then the game program will make the correct move in every situation and the game is said to be "solved" [Dre81].

Over the past two decades, great progress has been made in designing accurate position evaluators. Advances in computer game-playing have followed largely from the development of rigorous searching strategies which allow the evaluator to explicitly look ahead as many as 20 turns into the future, considering millions of possible game continuations to arrive at an excellent estimate of a position's expected payoff. Indeed, Connect-Four has been solved by this method [All89, All88], and deeply-searching chess and checkers programs now play at the world-class level [LN91, SCT+92]. Yet in the game of backgammon, significant advances have been made only very recently, and using an altogether different approach.

1.2.1 Backgammon

Backgammon, known variously as the "oldest" and the "cruelest" board game, defies solution by lookahead search because the number of possible moves at each turn is simply too large—over 400 on average, considering the 21 possible rolls of the dice. A program to play backgammon in real-time must therefore "not search more than one or two ply, but rather rely on pattern recognition and judgmental skills" [TS89]. The game of backgammon is a particularly well-suited domain for testing methods of noisy, stochastic, "real-life" pattern recognition.

The earliest serious attempt at a backgammon program was in the late 1970's, when Hans Berliner's group devoted four man-years to hand-crafting the "BKG" position evaluator [Ber79]. The evaluator made use of a large number of features F_i which indicated the presence or absence of various patterns of pieces on the board. The evaluator then combined these features into an overall payoff estimate according to Berliner's SNAC (Smoothness, Non-linearity, and Application Coefficients) methodology:

Smoothness The overall evaluation should define a smooth surface over the feature space. If the surface is discontinuous with respect to a feature in the space, then the program will tend to attach too much importance to that feature (and play incorrectly) in situations where alternate move choices are on opposite sides of the discontinuity.

Nonlinearity Linear functions, though well-behaved, are inadequate for modelling the context-sensitive relationships between features in the space; the evaluator should be able to combine the feature values nonlinearly.

Application Coefficients Berliner's solution to the smoothness and nonlinearity constraints was an evaluation function of the form:

$$V = A_1F_1 + A_2F_2 + \cdots + A_nF_n.$$

The A_i values, termed *application coefficients*, were special features whose values tend to vary slowly with respect to board moves. The A_i's indicate concepts such as the phase of game and ease of winning. Since these are slowly-varying, the overall evaluation will be relatively constrained from move to move, guaranteeing smoothness and stability while computing a nonlinear function of the discrete-valued features F_i.

In 1979, version 9.8 of the program actually defeated the world champion in a short match (although it was very lucky to have done so). According to backgammon expert Bill Robertie, BKG's expected payoff against a human expert would have been approximately -0.3 points per game (ppg), ranking it at the advanced intermediate level [Rob92]. No commercial backgammon programs developed before or since BKG have ranked higher than -0.66 ppg, testifying to the quality of Berliner's pioneering effort.

The BKG evaluator, a nonlinear function which smoothly combines many features of the input, naturally suggests an implementation as an artificial neural network. Indeed, backgammon-playing (along with speech recognition) was one of the first complex tasks to be attempted with an MLP and error back-propagation. In 1989, Tesauro and Sejnowski [TS89] trained a standard MLP on a database of 3202 positions labelled by a human expert. The network's optimal performance was achieved after only about 12 training cycles through the database, after which its performance slowly degraded. (By contrast, the network's performance on the training data was still steadily increasing after over 50 epochs of training, indicating that the networks were overlearning the noise in the training set.) The optimal network was able to win about 59% of its games against a weak computer opponent, and also defeated its rivals at the First Computer Olympiad [Tes89]. However, it made many obvious blunders in many different types of positions, and would have stood little chance of success against a good human player. Tesauro prophesised that improving the network further would require

> either an intractably large number of [training] examples, or a major overhaul in either the pre-computed features or the training paradigm. [TS89]

1.2.2 Reinforcement Learning

Tesauro was right. Several months ago, he unveiled his latest backgammon evaluator network, the result of a major overhaul in the training paradigm which allowed the net to see not just 3202 but literally millions of training examples—all generated from its own play [Tes92]. The new paradigm is based on the theory of reinforcement learning by the method of temporal differences (TD), formalised by Sutton [Sut88].

Temporal difference methods apply to the training of any system which learns to predict the outcome of a temporal sequence of events. Two examples of temporal prediction problems are speech recognition, viewed as a series of speech frames culminating in a phoneme classification, and game-playing, viewed as a series of board positions culminating in a win/loss decision. The distinguishing feature of TD methods is that the training is driven not by the difference between each prediction and the eventual outcome, but rather by the difference between successive predictions only.

More formally, suppose the prediction system has observed an *experience* consisting of a sequence of observation vectors

$$x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow \cdots \rightarrow x_{T-1} \rightarrow x_T$$

which culminated in the scalar outcome Z. The task of our system is to produce, at each timestep t, a prediction $P(x_t)$ estimating the outcome Z given the current observation; the task of the learning method is to modify the system's prediction function $P(x)$ in light of each new experience. In the traditional supervised-learning method, the experience above would generate the training pairs

$$(P'(x_0), Z), (P'(x_1), Z), \ldots, (P'(x_{T-1}), Z), (P'(x_T), Z)$$

adjusting each new prediction P' to more closely approximate the actual outcome of the experience. By contrast, the temporal difference method **TD(0)** would generate the training pairs

$$(P'(x_0), P(x_1)), (P'(x_1), P(x_2)), \ldots, (P'(x_{T-1}), P(x_N)), (P'(x_T), Z)$$

associating each observation vector with only the next timestep's prediction. TD(0), also known as Q-learning [Wat89], is actually the most extreme instance of the more general TD(λ) class of learning methods, with TD(1) = supervised-learning at the other extreme [Sut88].

Sutton provides the following example from game-playing to demonstrate how TD methods can generate better predictions than supervised-learning methods. Suppose the system observes a game that reaches a novel position x_t, then progresses to a position x_{t+1} which is known to be bad, but finally happens to result in a win nonetheless. In a supervised-learning system, the novel state would be associated with the final "win" outcome, in spite of its having immediately led to a bad position. In TD(0) learning, however, the novel state would be associated with its immediate

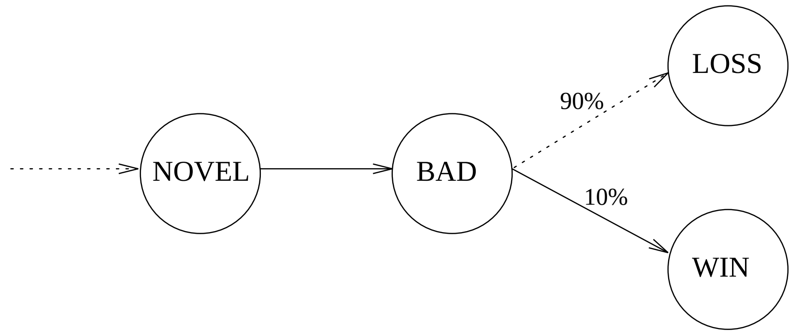

Figure 1-2: Supervised vs. TD(0) learning (from [Sut88]).

temporal successor, and the system would learn the correct, unfavourable prediction for that state. Because TD methods preserve the temporal structure of the experience in generating training data, they "make more efficient use of their experience, ...converge more rapidly and make more accurate predictions along the way" [Sut88].

Tesauro's newest backgammon position evaluator is a neural network trained by TD methods [Tes92]. The "experiences" used for generating training data were not human expert moves (as were used to train Tesauro's earlier backgammon network), but rather full games played by the network against itself. Thus, the network is used simultaneously as a predictor and controller: it predicts the expected game value given the board position, and it controls the progress of the game by selecting the highest-valued position reachable by a legal move at each step. Tesauro stresses that there are no theoretical guarantees that such a predictor/controller system will converge to an optimal or even good solution. In practice, however, the TD training scheme worked remarkably well on the backgammon task.

Tesauro trained a standard two-layer MLP consisting of a raw input board representation of 198 units, a hidden layer of 40 units, and an output layer of four units representing the probability of each possible outcome of a backgammon game.[1] After $200,000$ games of training by self-play, the network had learned enough backgammon strategy to win 50% of the time against Tesauro's previous best network, which was trained on expert play *and* had complex features encoded in the input representation. When Tesauro trained another TD network which included these complex features in its input, its performance shot beyond that of the human expert-trained networks. Expert Bill Robertie recently played it and found that, although it made "obvious technical errors" and "costly errors in complex positions," it was nevertheless "the strongest backgammon program in existence, most likely better than Berliner's program of 13 years ago" [Rob92]. Tesauro's initial results suggest that a connectionist evaluator trained by the method of temporal differences may eventually reach, or even surpass, human expert-level ability.

[1] A backgammon game actually has six possible outcomes (single, double, and triple wins or losses), but triple games are quite rare. With Tesauro's four-output representation, the overall value of the position is computed as $P(\text{win}) + 2P(\text{double win}) - P(\text{loss}) - 2P(\text{double loss})$.

Chapter 2

Experimental Design and Results

In backgammon as in every complex game, strategies change significantly as the game progresses. In other words, the game position evaluation function decomposes naturally into several simpler functions in different regions of its domain. Thus, it is reasonable to expect that a modular neural network could learn to approximate this evaluation function more accurately—and thereby play the game better—than a monolithic network could. This was the hypothesis which I proposed to explore for my summer project. Specifically, my goals were as follows:

1. Implement the algorithms for monolithic MLP back-propagation, temporal-difference learning, and game-playing;
2. Apply these algorithms to the task of learning tic-tac-toe and backgammon, comparing my results with those of Robinson and Fallside [RF89] and Tesauro [Tes92], respectively; and
3. Design and implement several modular neural network architectures for learning context-sensitive evaluation functions, and compare their performance to that of the monolithic networks.

2.1 Program Design

2.1.1 Implementation

The complete implementation of the network-training and game-playing algorithms comprises approximately 3000 lines of C++ code. C++ was chosen for its object-oriented features, which allow convenient interaction between the various modules of the program, as well as for its run-time speed. Figure 2-1 illustrates the interrelationships among the seven main program modules:

- GAMETRAIN provides the top-level interface between the user and the game-learning mechanism. According to a set of parameters specified on the command line, this module

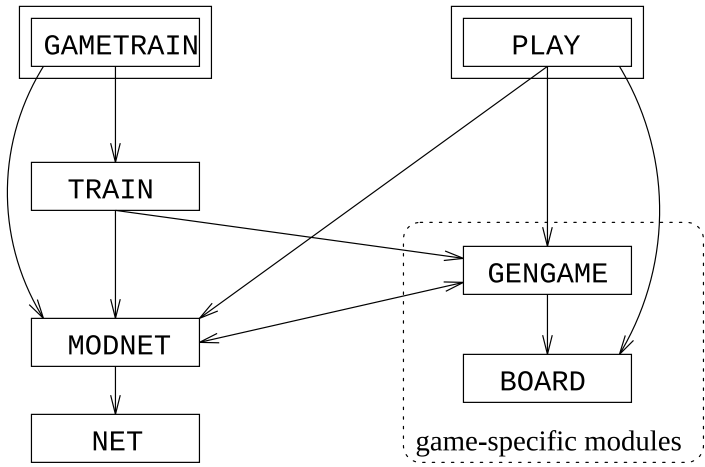

Figure 2-1: Organisation of C++ Program Modules

calls on MODNET to initialise or load a neural network, then uses TRAIN to carry out the learning process.

- TRAIN invokes GENGAME repeatedly to generate new games for use as training information. It segments the stream of game positions into training *epochs* consisting of 50 positions each, uses MODNET to actually train the net, and compiles per-epoch statistics on the training progress.

- GENGAME has three main functions:

  - To specify the encoding of the game position data structure into the neural network input representation;
  - To select best moves and generate full games using the neural network evaluation function; and
  - To provide a stream of training data for the network according to the TD(0) method of temporal differences.

  GENGAME is game-specific so there are actually two versions of this module: GENBG for backgammon and GENTTT for tic-tac-toe.

- BOARD defines the board representation, rules, board display, move notation, etc. for a specific game. Each BOARD module also includes a hand-written conventional evaluation function called HAND which can be used to measure the playing ability of the neural network evaluation function.

- MODNET defines the operations for evaluating, training, reading, and writing two kinds of modular neural networks. These modular architectures are described below in §2.1.3. MODNET relies heavily on the NET module, which handles the training and evaluation of the modular net's monolithic subnetworks. It may also call the game-specific gating program in GENGAME to determine which subnetwork applies to a given position.

- NET implements the forward-propagation and backward-propagation algorithms for a multilayer perceptron [RHW86], as well as providing utility routines for reading, writing and displaying the contents of such networks.

- PLAY provides a general interface for playing two-player games. On the command line, the user specifies who will play each side (either human, Hand, or a network) and the number of games to be played. The games may be played in verbose mode, in which case the board is displayed after each move, or in quiet mode, in which case the output is simply a number representing the average number of points per game won by player X.

The networks were trained on a DecStation 3100 MIPS machine. The playing code was also ported to an AST 486 PC in order for the backgammon program to compete in the 1992 Computer Games Olympiad.

2.1.2 Details of Neural Network Training

Designing an MLP neural network requires the researcher to define a number of parameters such as the number of hidden layers, number of units in each layer, squashing function, learning rate, and momentum coefficient [RHW86]. In order to establish these, I experimented with different network parameter settings for learning three small tasks:

exclusive-or This is the classical test of an MLP's ability to learn a nonlinear function.

2-D Euclidean distance from origin This task tests the network's ability to approximate an analog function accurately.

Bounded random walk This problem, described in [Sut88], tests the ability of a network to solve a linear prediction problem by the method of temporal differences.

Based on the results of these experiments, I set the network parameters as described below. Note, however, that it was not a primary goal of my project to fine-tune the network parameters for the fastest learning rate possible. The following settings seemed to work well:

- Each network had one layer of hidden units, squashed by the sigmoid function $\sigma(x) = 1/(1+e^{-x})$, and with a maximum learning rate of $\frac{1.0}{\#\text{ of input units}}$. The actual learning rate used during training was the product of this maximum rate with the adaptive learning rate multiplier η_x (described below).

- *No* squashing function was applied at the output units, which received activation directly from the input units (via *shortcut connections*) as well as from the hidden units. The shortcut links used a maximum learning rate of $\frac{0.005}{\#\text{ of input units}}$, and the hidden-to-output links used a maximum learning rate of $\frac{0.005}{\#\text{ of hidden units}}$.

- Network weights were initialised randomly within the range $(-0.2, +0.2)$.
- I designed the following heuristic scheme for adapting the learning rate multiplier η_x and momentum coefficient μ after each epoch of training:
  - If the log-mean-sum-squared-error (LMSSE) has decreased by δ since the previous training epoch, then raise μ and lower η_x according to

$$\mu \leftarrow \tfrac{\mu+\delta}{1+\delta}; \quad \eta_x \leftarrow \tfrac{\eta_x}{1+\delta}$$

  - If the LMSSE has *increased* by δ since the previous training epoch, then lower μ and raise η_x according to

$$\mu \leftarrow \tfrac{\mu}{2}; \quad \eta_x \leftarrow \tfrac{\eta_x+\delta}{1+\delta}$$

- The back-propagation algorithm updated the network weights after each training pattern, rather than only once per epoch. As suggested in [Fah88], the constant 0.1 was added to the sigmoid derivative in order to keep the hidden units from getting prematurely "stuck" near 0 or 1.
- I applied a method similar to that used in [HWSS88] for skipping a training pattern when the network error on that pattern was already low. Specifically, if the pattern's log SSE was less than the last epoch's log mean SSE by at least 3.0, then no backpropagation was done on that pattern. This scheme offers two advantages: it saves the extra time which would have been used to backpropagate very small weight changes, and it helps prevent over-training in cases where the training patterns are not distributed evenly over the input domain.

2.1.3 Design of Modular Architectures

The MODNET program module implements the algorithms for two types of modular neural network architecture. The first of these I call a DDD network, which stands for Designer Domain Decomposition (see figure 2-2). The DDD network consists of a collection of N monolithic subnetworks and a hard-coded gating function, written by the designer, which partitions the domain space into N classes. Like the all-consonant recognition network described in [WSS89], the DDD net allows the designer to use his or her domain knowledge in specifying a useful decomposition. The operation of the DDD net is extremely simple: in both forward and backward propagation, the gating program is called to select an appropriate subnetwork, which is then evaluated or trained as a monolithic net. Exactly one subnetwork is active at any time; for example, in figure 2-2, the gating program classifies the input pattern as belonging to class #3, so only subnetwork #3 is used to generate the modular network's output. Thus, assuming that the gating program can classify an input pattern in a negligible amount of time, the DDD net runs as quickly as a monolithic network the size of just one of its subnetworks.

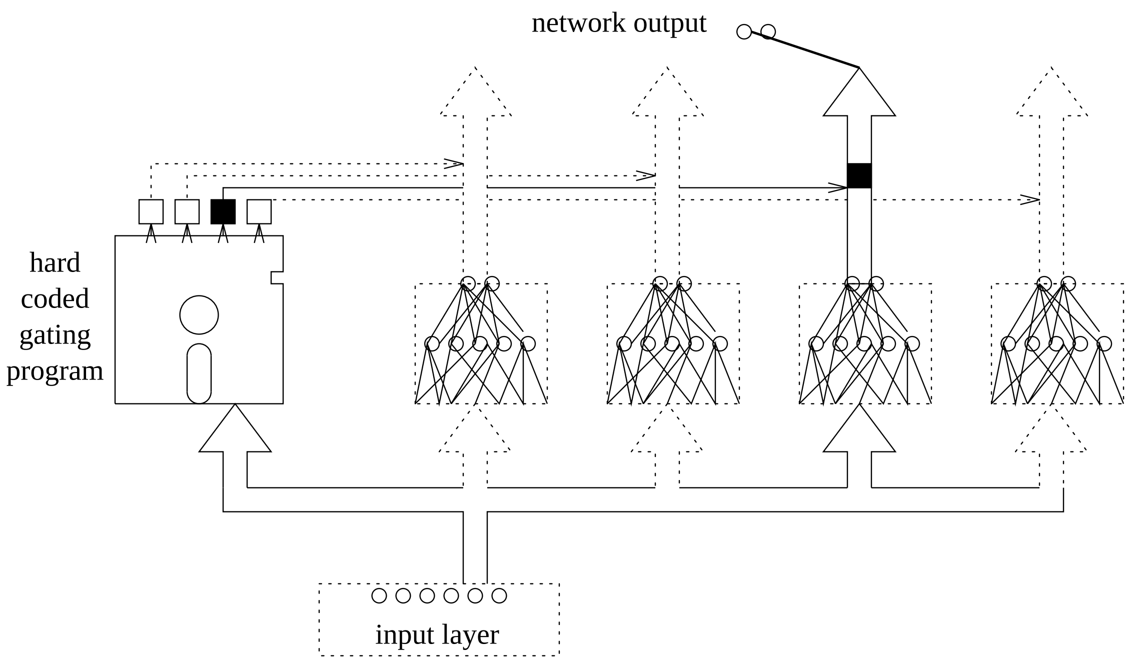

Figure 2-2: The DDD Modular Network

The second type of modular network implemented in MODNET is the Meta-Pi architecture [HW89]. The structure of the Meta-Pi network is very similar to that of the DDD net, with the important exception that the hard coded gating program is replaced by a trainable gating network (see figure 2-3). The weight changes for the gating network are computed by backpropagating the following error derivative from each output M_k:

$$\frac{\partial E}{\partial M_k} = \sum_N \left((O_n - D_n)(\rho_{k,n} - O_n)\right) / \sum_J M_j$$

where O_n is the nth output of the whole network, D_n is the desired target value, and $\rho_{k,n}$ is the nth output of subnetwork k.

In training a Meta-Pi net, the subnetworks are assumed to have already been fully trained; only the gating network is modified. Thus, the procedure I followed was first to design and train a DDD network, then replace the hard-coded gating program with the Meta-Pi gating network and re-train. Intuitively, it may seem that the Meta-Pi network, initialised in this way, can never do better than to learn to produce exactly the "hard" classification decisions defined by the DDD net's gating program, and thus can never exceed the performance of the original DDD net. On the other hand, the "soft" decisions made by the Meta-Pi gating network provide the advantage of combining the outputs of the subnetworks *smoothly*. Recall Berliner's smoothness criterion for evaluation functions: if the function is not smooth, then

> a very small change in the value of some feature could produce a substantial change

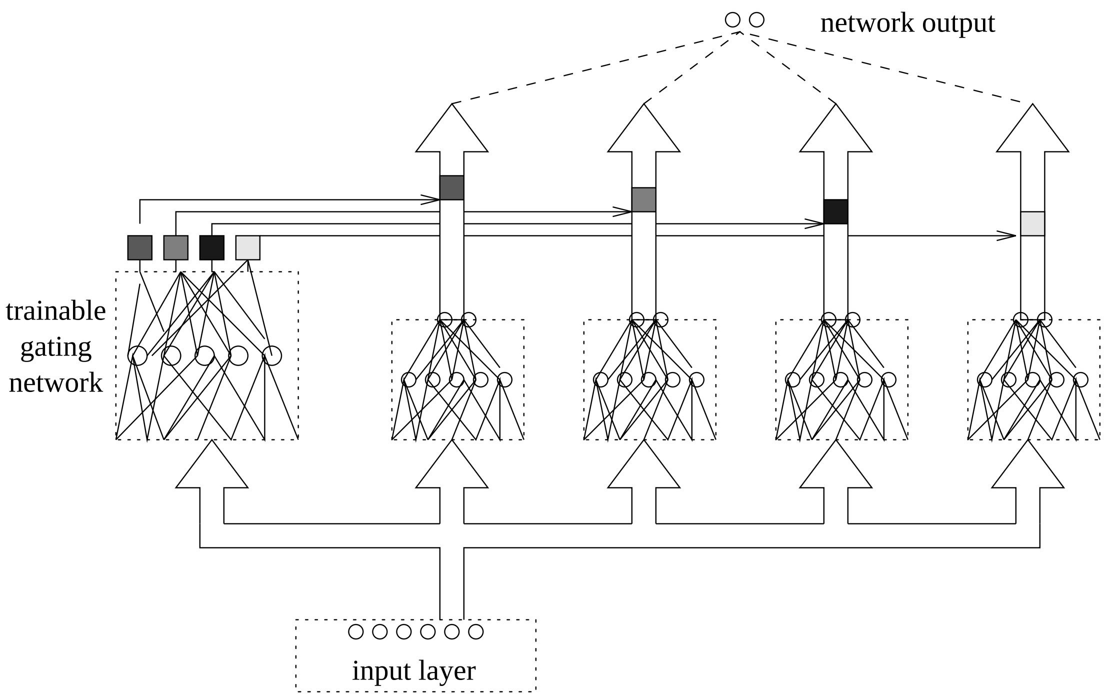

Figure 2-3: The Meta-Pi Network [HW89]

> in the value of the function. When the program has the ability to manipulate such a feature, it will frequently do so to its own detriment. [Ber79]

For example, suppose our evaluation function is represented by a DDD network and that the program has to choose between a legal move in domain class #3 and a legal move in domain class #5. If network #3 has a positive bias and network #5 has a negative bias with respect to the ideal evaluation function, then our program will tend to leap prematurely to the positively-biased region of the domain, #3. Berliner labelled this problem **the blemish effect**.

The soft gating function of the Meta-Pi network can smooth out the evaluation function in the transitional regions between the subnetwork domains. In effect, the gating network learns to produce output coefficients which are precisely analogous to the Application Coefficients of Berliner's 1979 backgammon program: they are slowly-varying features of the input which smooth the transitions from one region of the evaluation function surface to the next. Ideally, one would hope that the Meta-Pi network could *learn* a backgammon evaluation function of a similar form to, and of a higher quality than, the excellent evaluation function hand-crafted by Berliner's team 13 years ago.

2.2 Tic-Tac-Toe

Before attacking the complexities of backgammon, I applied the combination of neural networks and temporal-difference learning to the game of tic-tac-toe, also known as noughts-and-crosses.

Tic-tac-toe is a conceptually simple game with a small state space (on the order of 10^4 reachable positions), yet the optimal static position evaluator must be a complex nonlinear function of the input position in order to recognise such features as the potential for forking plays.

2.2.1 Design

Since tic-tac-toe is deterministic, a program that is learning by self-play may become stuck in a local minimum where it plays the same game repeatedly to a draw but has failed to explore large regions of the input space. In order to force the network into situations where it can learn about new ways to win the game, the self-play training procedure must involve some nondeterminism. An alternative approach is to train the network on games played against a high-quality opponent algorithm. This algorithm would be able to "teach" the network by leading it into new regions of the game position space. This is the method I used to train my tic-tac-toe networks.

The network's training opponent, T-HAND, plays a reasonable game of tic-tac-toe by following this simple algorithm: on each turn, always complete a line if possible; else always block the opponent if she is about to complete a line; else make a legal move at random. I estimate that an optimal player could expect to win 58% of the time and draw the remaining 42% against T-HAND.[1] The network training data was generated from the games played by the TD(0) reinforcement learning method. As in [RF89], no attempt was made to take advantage of the symmetries of the game.

Two tic-tac-toe evaluator networks were trained: a monolithic network and a modular network. The monolithic network was a standard MLP (as described above in §2.1.2) with 18 binary input units, 20 hidden units, and a single output unit. The modular network was a DDD net with 8 subnetworks, each having the same 18-20-1 architecture as the monolithic network. The DDD gating program assigned a position with N pieces on it ($1 \leq N \leq 8$) to subnetwork # N; e.g., subnetwork #6 was responsible only for evaluating tic-tac-toe positions with six total marks on the board. This particular decomposition of the domain has the advantage that all legal moves at each turn are evaluated by the same subnetwork. Thus, the program does not have the ability to manipulate "to its own detriment" [Ber79] the subnetwork it uses for a given move, so there is no need to use a Meta-Pi network to smooth the overall evaluation function.

2.2.2 Results

The networks trained against the T-HAND algorithm learned to play an extremely strong game of tic-tac-toe. Figure 2-4 compares the learning curves of the monolithic and modular networks

[1] When the optimal player goes first, she wins 7/8 of the time by starting in a corner, forcing T-HAND's random move selector to select the center square immediately to avoid being forked two turns later. When T-HAND goes first, the optimal player must be both clever and lucky to be able to set up a winning forking play; she can expect to do this in 92/315 of these games. So the overall expected equity is the average of 7/8 and 92/315, which is about +0.583.

trained in this manner. On this graph, each range of ten units on the x-axis corresponds to approximately 70,000 training games, or one hour on a DEC 3100 workstation. The monolithic

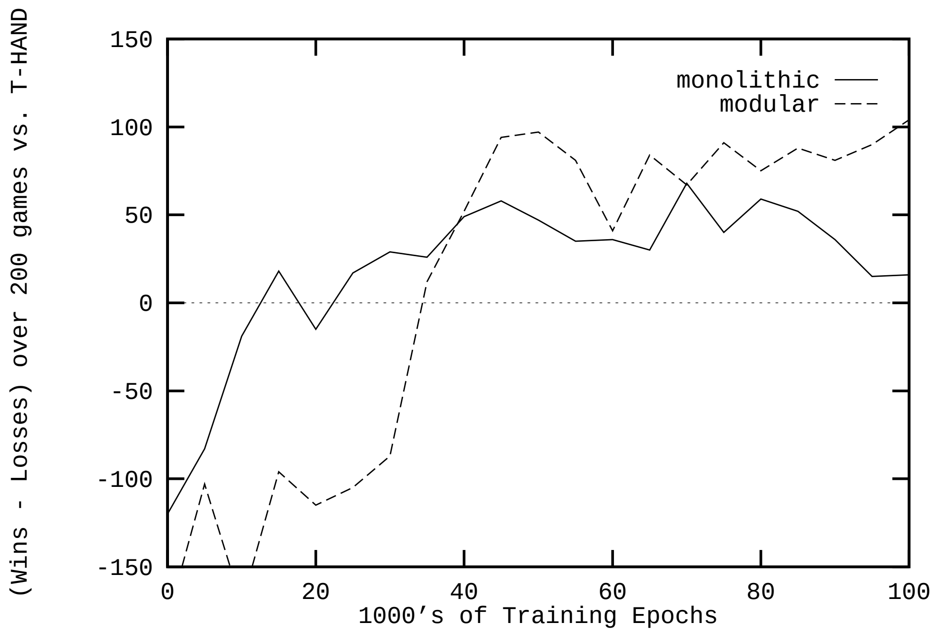

Figure 2-4: Performance of tic-tac-toe networks trained against T-HAND

network learned more quickly at first than its modular counterpart, no doubt because each modular subnetwork saw only 9%–14% as many training patterns per epoch as the monolithic net did. However, the performance of the modular network soon surpassed that of the monolithic network, and indeed approached the optimum expected equity of 0.58 (or 116 wins out of 200). In a match of 10,000 games, the best modular network won 4780 and lost only 45, for an average equity of +0.47 (see table 2.1). These results compare very favourably with those reported by Robinson and Fallside [RF89], whose best tic-tac-toe network attained an equity of +0.18 against an opponent algorithm considerably weaker than T-HAND.

Network Type	Training	Wins	Draws	Losses	Equity
monolithic	vs. T-HAND	3502	5976	522	+0.2980
modular	vs. T-HAND	4780	5175	45	+0.4735

Table 2.1: Performance of best networks in 10,000 games vs. T-HAND

2.3 Backgammon

Unlike tic-tac-toe, backgammon has several aspects that suit it particularly well for learning by *self-play*:

- The game begins in a unique starting state and proceeds stochastically through a transition network of states each of which conveys *perfect* (i.e. complete) *information* about the likely eventual outcome;
- One player is guaranteed to win eventually even if both sides play randomly, thus each game is guaranteed to produce a useful nonzero reinforcement signal;
- The nondeterministic element introduced by the dice rolls forces the learning system into many different types of positions without the need for explicit "exploration"; and
- Backgammon's aesthetic nature means that in general, good strategies build on each other smoothly, i.e. they may be discovered incrementally. By contrast, strategies for other games (say, a chess endgame problem) offer more of an all-or-nothing reinforcement signal, which would reward the learning system only after it had managed to discover a complicated sequence of positions.

The first two aspects identify backgammon as a *Markov decision problem* to which reinforcement learning techniques may conveniently be applied, and the last two aspects largely relieve the learning system of the need to attempt risky and time-consuming techniques for exploring the space [Thr92].

2.3.1 Monolithic Networks

2.3.1.1 Design

I wanted to test backgammon learning both from games generated by self-play and from games played against a good conventional algorithm. In backgammon, however, it is no trivial task to write a high-quality opponent. My algorithm, B-HAND, knows only a minimum of basic backgammon heuristics such as "make inner board points", "hit your opponent", and "try not to get hit". Although it beats a random player 90% of the time, it could almost never win against a skilled human player, and indeed would often lose bonus points (called "gammons"). Nevertheless, between the two extremes of a random player and a skilled human, the B-HAND algorithm gives us a useful yardstick for measuring the ability of its opponents.

For both the "versus B-HAND" and self-play training paradigms, I trained MLP's (configured as described in §2.1.2) with 0, 20, and 50 hidden units, respectively. The input layer to all the networks was somewhat more compact than Tesauro's 198-input network [Tes92], with 156 units distributed as follows:

- For each of the 25 "points" on a backgammon board, six binary units encoded whether there was one X piece, two X pieces, three or more X pieces, one O piece, two O pieces, or three or more O pieces on that point. (Tesauro's network truncated at four rather than at three pieces, explicitly encoding the number of pieces beyond three with an additional analog unit for each point.)

- Two analog units encoded each side's "pipcount", a ubiquitous statistic in backgammon which measures the total distance all of a player's pieces must travel to reach the end of the board.[2] The pipcount is in fact a linear feature of the complete raw board position, but because of my truncated position encoding, it provides important information about the game which could not be deduced from the other input units.

- Two analog units encoded the percentage of pieces each side has removed from the board, and two more binary units encoded whether *any* pieces have yet been removed from the board. These statistics are very important in the endgame for determining how close each side is to winning and whether there is a possibility of winning bonus points.

- Unlike Tesauro's network, my nets used no input units to identify the player whose turn it was to move. Rather, the network always saw positions in which it was player O's turn to move; when it was X's turn the board position was simply reflected and the X and O labels swapped.

This representation is "raw" in that the input units are set directly from the board position without any involved calculation. On the other hand, the representation does take advantage of a bit of the designer's knowledge of backgammon strategy—for example, the knowledge that it is useful to have a separate representation for the cases of one, two or three pieces on a point, but not usually important to distinguish among cases with more than three pieces on a point. The goal is to provide a representation which is compact enough that learning can proceed quickly, yet full enough to allow the network to discover relevant higher-order features of the position.

The networks have just two output units—again a more compact representation than Tesauro's nets, which used four outputs. The first of these predicts the outcome of the game ($+1$ for a sure win by X, down to -1 for a sure win for O) regardless of possible bonus wins. The second unit predicts the expected number of bonus points (for example, $+1$ for a likely X gammon, or -2 for a likely backgammon by O). A weakness of this output representation is its inability to distinguish between "mutually gammonish" positions—in which both sides have a good chance of winning bonus points—and positions in which neither side is likely to win a gammon. This distinction could influence move selection during matches played to a fixed number of points.[3] However,

[2]Since each player's pipcount in the starting position is 167, the pipcount is scaled by 1/167 before being input to the network.

[3]It would also be important for making good "doubling cube" decisions during match play. However, like Tesauro, I ignored backgammon's doubling cube option since including it would have complicated the payoff calculation considerably.

the more compact network does have fewer weights and thus should train more quickly than Tesauro's larger networks. Using this representation, the overall game expectation for player X is given simply by the sum of the two output activations.

2.3.1.2 Results

Training the monolithic neural networks by TD(0) on the game of backgammon produced contrasting results: the self-trained networks learned to play impressively well, while the networks trained by playing games against B-HAND made only slight progress and were never able to beat B-HAND consistently. Figure 2-5 shows the learning curves of the three monolithic architectures trained against B-HAND. In this and all following graphs, each unit on the X-axis represents an epoch of 50 training positions, or approximately one game; the Y-axis represents the average number of points per game won by the network in a 100-game match against B-HAND. The margin of error at each data point is approximately 0.15 ppg.

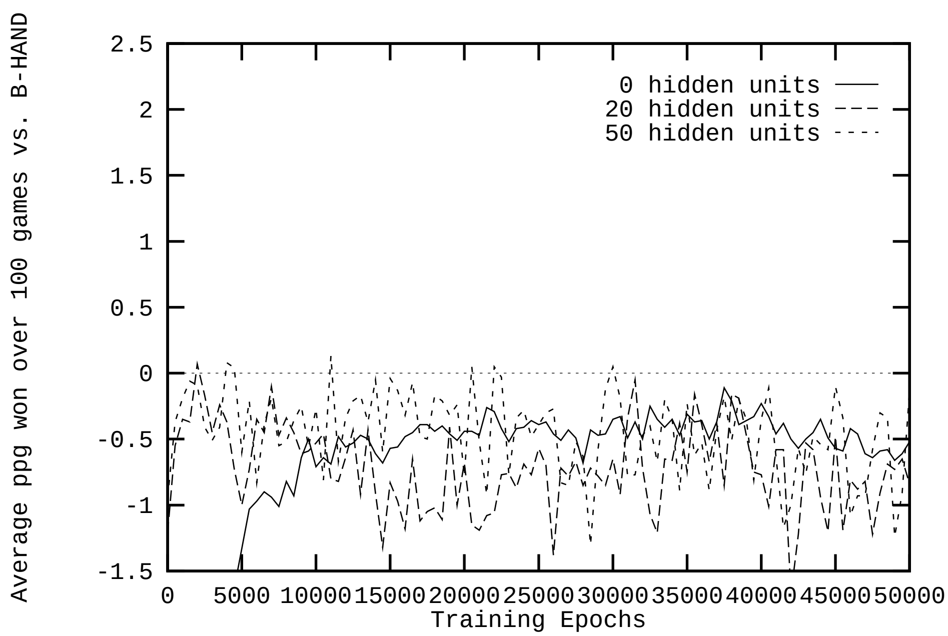

Figure 2-5: Performance of backgammon networks trained against B-HAND

These networks, although they improved substantially over their initial (random) move selection, did not achieve excellent performance because of the weakness of the opponent algorithm they trained against. To see how a weak training opponent can stall learning, consider tic-tac-toe position P shown in figure 2-6. Suppose the teaching algorithm, playing X, is weak and fails to make the winning play in position P. In this case, the network will not learn to associate P with winning, and thus it will learn neither how to reach nor how to defend against P. Even if the network does stumble upon the winning play from position P on its own, the positive

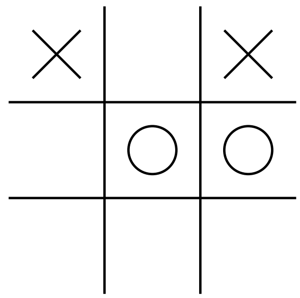

Figure 2-6: Effect of a Weak Training Opponent. X to Play

reinforcement signal to P would only be cancelled by the negative reinforcement generated the next time its opponent misplayed the same position.

My backgammon algorithm B-HAND is evidently weak enough to mislead the networks in this way. A possible remedy would be to train the network on only half of the game positions—namely, those in which the network had just moved. The network would then see only positions from its own perspective and learn only about the effects of its own actions. On the other hand, this strategy would result in a network specifically fine-tuned to take advantage of its opponent's weaknesses. I did not have time to test this remedy, and in any event, for backgammon the success of the learning by self-play paradigm would appear to make hand-written teaching algorithms superfluous.

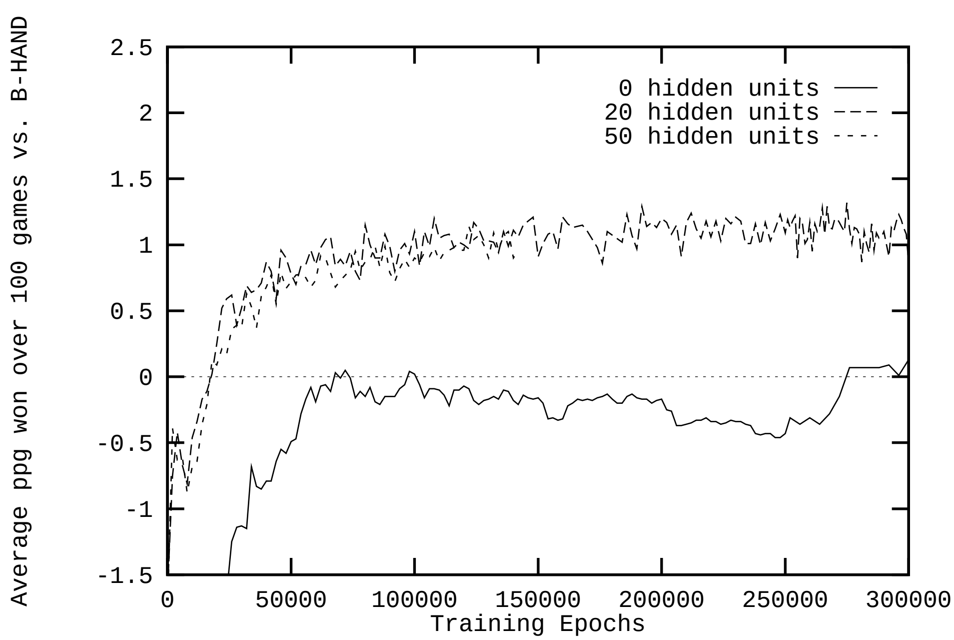

Figure 2-7: Performance of monolithic backgammon networks trained by self-play

In learning from self-play, the problems of a poor teacher are eliminated because each side agrees with the choices made by the other; thus, there are no conflicting behaviours to muddle the propagation of the reinforcement signal. My monolithic networks trained by self-play were able to learn an impressive amount of backgammon strategy: after 80,000 training epochs, the 20-hidden-unit network was able to beat B-HAND 8 times out of 10—a tremendous advantage in backgammon. The learning curves (see figure 2-7) show that the networks with hidden units reached an expected value of more than 1.0 points per game, reflecting the substantial number of bonus points won against B-HAND. Even the linear perceptron with 0 hidden units was able to learn enough strategy to play as well as the hand-written algorithm. These results provide independent confirmation of Tesauro's successful application of TD methods to backgammon learning [Tes92].

What the graph of figure 2-7 does not reflect is the dramatically different training times required for the three network topologies. On the DEC workstations, the amount of time required to perform 5000 training epochs was approximately one hour for the 0 hidden unit network, ten hours for the 20 hidden unit network, and one day for the 50 hidden unit network. In fact, the best 20-hidden network not only learned faster but also played slightly better than the best 50-hidden network: in a 500-game match played between them, the 20-hidden network won by a total point score of 380 to 335 (see table 2.2, page 27 for details). However, with further weeks of training, the 50-hidden network might be able to take advantage of its greater information capacity to surpass the smaller net's playing ability.

The best monolithic networks play a good game of backgammon, demonstrating an excellent ability to recognise important patterns of pieces on the board and evaluate the strategic trade-offs involved. As Tesauro found, they are especially good in the most frequently-encountered situations such as running, holding, and blitzing games. However, my networks err badly in certain positions for which the normal rules of thumb do not apply. For example, they will continue to attack their opponent's pieces agressively even if the opponent has built a "prime" formation which makes such attacks futile. They also play weakly in the endgame, after contact is no longer possible. In this phase of the game, the normal rules of thumb such as "maintain points" and "don't leave blots" are totally irrelevant; yet the network's play still seems biased by these heuristics. Learning proper endgame play is also difficult because the plays are often very close: more often than not, the precise move chosen will *not* affect the outcome of the game.

For example, consider the endgame position of figure 2-8, in which O is well ahead in the race and will win easily. X, to play the dice roll of [5]-[4], should bring his furthest pieces around towards his inner board (points 1–6) so as to avoid losing bonus points. However, the monolithic network chooses to make its 3-point (`8/3; 7/3`) in this situation, a serious error. Making the 3-point would have been correct if X still had a chance of contact with O; however, in the context of this racing game, the 3-point has no value whatsoever. In my opinion, the correct play here is either `15/6` or `21/12`, but these were ranked 9th and 15th out of the 30 possible moves. This example typifies the weakness of the monolithic networks in contexts which require the network's usual heuristics (e.g., *make the 3-point*) to be violated. It was my hope that a modular neural network could provide the added context-sensitivity needed to play such moves correctly.

2.3.2 Modular Networks

2.3.2.1 Design of DDD Nets

With the Designer Domain Decomposition architecture (figure 2-2, page 14), the network may learn easily that a given pattern of pieces is important in one context but irrelevant in another—so long as the gating program assigns those two contexts to different subnetworks. The better our gating program is at isolating regions requiring context-sensitive evaluations, the better the quality of learning we can expect. On the other hand, if we partition the space into too many separate classes, then each subnetwork will see proportionately fewer training examples, and the learning process will be slower. For classes which require evaluation functions that are quite similar, the hidden units of an MLP should provide enough context-sensitivity to distinguish between them, so they should be treated as the same class for the purposes of designing a DDD network.

For a game evaluation function, an effective gating program should consider the board position and try to classify it according to each side's current best strategy and the phase of game, much as human players classify positions. For example, classes of backgammon positions include "saving the gammon" (as in figure 2-8), "holding", "running", "blitzing", "prime vs. prime", and "back game" positions. However, the definitions of these classes are fuzzy at best, and usually several different strategies are all relevant to evaluating a position. For my experiments, rather than struggling to design the best possible partitioning of the space of backgammon positions, I opted for a 12-class scheme based only on the current race statistics ("pipcounts"):

- Nine of the classes model contact positions. X's and O's pipcounts are separately quantised as large, average, or small; this defines $3 \times 3 = 9$ distinct regions of the domain.[4]

- The other three classes represent non-contact positions, separating the cases where O has a large racing lead ($\geq$ 24 pips), X has a large racing lead ($\geq$ 32 pips), or the race is relatively close.[5]

As with the encoding of the raw board position into the input units, I again used only a minimum of domain knowledge to structure the learning network. Backgammon players recognise that this modularisation overemphasises the importance of the racing situation, and that the value of "deeper" structural features will need to be discovered independently by many of the 12 subnetworks. I also did not attempt to match the architecture of each subnetwork to

[4]Because the network always evaluates positions from X's perspective with O to play, we cannot use symmetry to collapse together opposite classes, e.g. the "X-large O-small" class and the "X-small O-large" class.

[5]The discrepancy between 24 and 32 accounts for the fact that O will gain an average of 8 pips on his next dice roll.

```
    24    23    22    21    20    19  bar   18    17    16    15    14    13
  /--v-----v-----v-----v-----v-----v---+-+---v-----v-----v-----v-----v-----v--\
# | [O]   [O]   [O]   <X>              | |                    <X>             |
# | [O]   [O]   [O]                    | |                                    |
  |       [O]                          | |                                    |
  |       [O]                          | |                                    |
  |       [O]                          | |                                    |
  |                                    | |                                    |
  |                                    | |                                    |
  |                                    | |                                    |
  |                                    | |                                    |
  |                               <X>  | |                                    |
  |                         <X>   <X>  | |                                    |
  |                         <X>   <X>  | |                                    |
  |                         <X>   <X>  | |  <X>   <X>                         |
  |                         <X>   <X>  | |  <X>   <X>                         |
  \--^-----^-----^-----^-----^-----^---+-+---^-----^-----^-----^-----^-----^--/
 0   1     2     3     4     5     6         7     8     9    10    11    12

   Ranking of legal plays for [5]-[4] by bgselfnet.20.251000:
       (1) : 8/3; 7/3        -1.90      <--- selected play
       (2) : 15/10; 10/6     -1.93
       (3) : 15/10; 8/4      -1.93
       (4) : 8/4; 7/2        -1.94
       (5) : 7/2; 6/2        -1.94
       ... 25 more ...
```

Figure 2-8: Poor Play Due to Context-Insensitivity of Monolithic Network

```
   (a) DDD Network

   Ranking of legal plays for [5]-[4] by pipnet9.hd.250000:
       (1) : 21/16; 16/12    -1.81      <--- selected play
       (2) : 21/16; 8/4      -1.82
       (3) : 15/10; 8/4      -1.82
       (4) : 15/10; 10/6     -1.84
       (5) : 21/17; 15/10    -1.85
```

```
   (b) Meta-Pi Network

   Ranking of legal plays for [5]-[4] by mp5net9.hd:
       (1) : 15/10; 10/6     -1.59      <--- selected play
       (2) : 21/16; 16/12    -1.64
       (3) : 21/17; 15/10    -1.65
       (4) : 15/10; 8/4      -1.66
       (5) : 21/16; 15/11    -1.67
```

Figure 2-9: Good Play by the Context-Sensitive Modular Networks the structure of its particular subproblem, as was shown effective in [JJB91]. Nevertheless, the modular network does provide a greater degree of context-sensitivity than the monolithic one. When Bill Robertie judged Tesauro's monolithic backgammon network, he observed that

> in backgammon positions (including back games) in which the race has gotten longer than in the starting position ... the race is relatively less important, and priming points are much more important. These positions arise relatively rarely, and as a result TD-Gammon hasn't yet learned that it needs a *different evaluation function* for these complex positions. [Rob92] (emphasis added)

My DDD modular network does in fact provide different subnetworks, and hence different evaluation functions, for unusual racing situations.

Another implementation decision concerned how the 12 DDD subnetworks would be initialised. Should they be given random weights (as the tic-tac-toe networks were), or should they all be initially cloned from a good monolithic network? The latter method gives the modular network the benefits of the monolithic network's faster early training, but on the other hand, it may be difficult for each subnetwork to re-assign its hidden units to improve accuracy on its new, more specialised task. I tested both initialisation methods and two different subnetwork topologies, making four DDD backgammon networks in all:

- 0 hidden units per subnetwork (i.e., 12 linear perceptrons), each initialised randomly;
- 0 hidden units per subnetwork, each initialised as a clone of a linear perceptron trained for 250,000 epochs;
- 20 hidden units per subnetwork, each initialised randomly; and
- 20 hidden units per subnetwork, each initialised as a clone of a monolithic MLP trained for 80,000 epochs.

2.3.2.2 DDD Network Results

Figures 2-10 and 2-11 show the learning curves of the DDD networks with 0 and 20 hidden units, respectively. In both cases, the modular networks learned to play a strong game of backgammon, overwhelming the B-HAND algorithm by approximately the same margins as had their monolithic counterparts. The nets whose component subnetworks were initialised as clones of a monolithic network appeared to learn more quickly and slightly better than those whose subnetworks were trained from scratch, although further independent training runs would have to be conducted in order to establish this conclusively.

It surprised me that the modular networks with 0 hidden units were unable to do better against B-HAND than the monolithic perceptron had done. After all, the monolithic perceptron with no hidden units can provide *no* context-sensitivity whatsoever for the evaluation function,

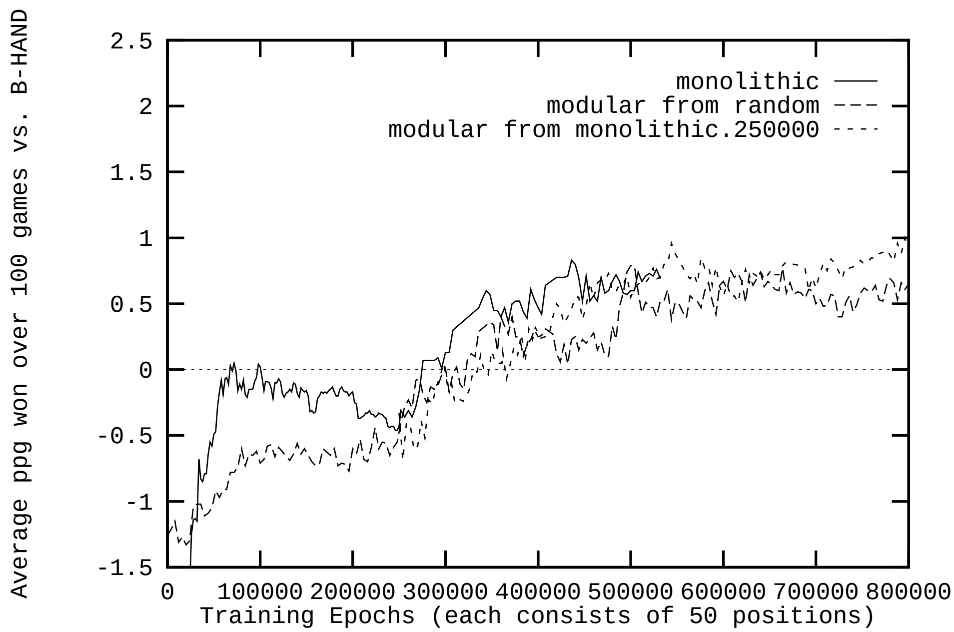

Figure 2-10: Performance of DDD Modular Backgammon Networks, 0 hidden units

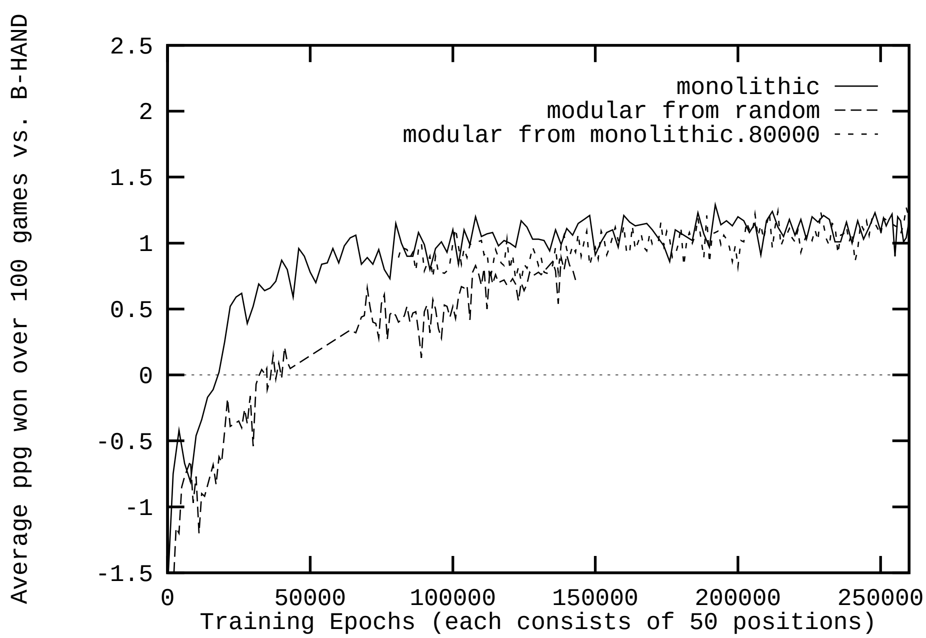

Figure 2-11: Performance of DDD Modular Backgammon Networks, 20 hidden units

while the modular net can at least change its evaluation function based on the racing situation. To see which network was really the better backgammon evaluator, I pitted the two nets against one another in a match of 500 games. The outcome was that the modular network soundly defeated the perceptron, winning an average of +0.35 points per game (see table 2.2). So how can we explain the perceptron's success against B-HAND? The answer lies in the fact that the B-HAND evaluator is itself largely a linear function of the board position. Because B-HAND's structure is similar to that of the linear perceptron, the perceptron (trained by self-play) may actually anticipate B-HAND's moves *better* than the ideal evaluation function, which expects its opponent to play optimally! In effect, the linear perceptron takes advantage of a "user model" of B-HAND's style of play. This user model does not help it, however, in head-to-head play against a decent opponent. The upshot of this discussion is that we should be wary of using results against B-HAND to draw fine comparisons between other networks.

Player X hidden, epochs	Player O hidden, epochs	X wins games	 bonus points	O wins games	 bonus points	X equity (avg. ppg)
monolithic 20, 284K	monolithic 0, 500K	370	234	130	47	+0.854
monolithic 20, 284K	monolithic 50, 132K	265	115	235	100	+0.090
modular DDD 0, 856K	monolithic 0, 436K	301	197	199	124	+0.350
modular DDD 20, 250K	monolithic 20, 284K	274	114	226	75	+0.174
modular DDD 20, 259K	monolithic 20, 265K	287	115	213	81	+0.216
DDD 2-ply 20, 250K	DDD 1-ply 20, 250K	30	11	20	8	+0.26
Meta-Pi 0, 824K	DDD 0, 800K	58	35	62	32	+0.01
Meta-Pi 20, 178K	DDD 20, 171K	40	16	40	22	+0.07

Table 2.2: Results of head-to-head backgammon matches

As opposed to the linear perceptron, the monolithic network with 20 hidden units does have the ability to model a nonlinear, context-sensitive evaluation function. Thus, we might expect a modular network to produce only slight gains over the monolithic MLP. In fact, though, the gains were substantial: head-to-head matches between the best monolithic and best modular networks consistently favoured the modular networks by about 0.2 points per game (see table 2.2). The modular networks did especially well in the "bonus" column, usually winning about 50% more bonus points than their monolithic opponents. These statistics indicate that the modular networks have indeed improved their play in complex situations with uneven pipcounts—precisely the type of situation which tends to lead to the award of bonus points. For a concrete example, in the endgame position discussed above (figure 2-8, page 24), the modular network does choose a good play (21/12) for the current context, and it ranks the monolithic network's poor choice (8/3; 7/3) 23rd out of 30 possible moves. The top alternate move choices of the modular network (see figure 2-9a) clearly indicate that the network has learned the proper strategy for this endgame context. In sum, the experiment of modularising the networks to increase their context-sensitivity was a success.

2.3.2.3 Competition Results: The Computer Games Olympiad

From August 5–8, 1992, my best modular networks competed in the backgammon division of the Fourth Computer Games Olympiad. The program, which I called MAESTRO 1.0[6], faced two conventional non-learning backgammon program opponents—VideoGammon and BAX. For the competition, I modified the playing algorithm in two ways:

- I allowed the program to search to the 2-ply level to resolve close calls among the network's top five 1-ply move choices. In other words, for each of the top five moves, the program would generate the opponent's best response to that move for each possible dice roll. The values of these 21 response plays were averaged to determine the ranking of the current position. This modification, though computationally expensive[7], improved the network's playing ability considerably (see table 2.2, page 27). Qualitatively, the network's judgment concerning when to take risks and when to play safely seemed very much improved. The extra ply of lookahead helped the network make up for its lack of pre-computed input features such as the risk of being hit.

- I implemented a rudimentary doubling cube strategy based on the current match score and the network's estimate of the position equity. In fact, MAESTRO's doubling behaviour was rather too optimistic: five times against BAX it lost while holding a cube at value 4, and the deciding game of the final match had a value of 8 points.

MAESTRO began the competition as the 20 hidden unit DDD network with 234,000 training epochs. During the first two days, this network beat VideoGammon 17–15 but lost to BAX by a score of 17–11. On the final day, MAESTRO was updated to the most recent network, which had been trained for 250,000 epochs; it beat VideoGammon 13–3, but lost to BAX 17–13, thus finishing in second place in the tournament. All three programs played at a solid intermediate level by human standards, with VideoGammon perhaps slightly weaker than MAESTRO and BAX. The other competitors and I agreed that with backgammon's strong chance element, a

[6] "M" for Modular, "AESTRO" for Tesauro (almost!).

[7] Move selection took between 15 and 45 seconds on an AST 486 PC.

longer series of matches would be needed to determine which of the programs was truly strongest. There is no doubt, however, that a human expert could have defeated all three programs. For further details of the Olympiad results, please refer to [Boy93].

2.3.2.4 Meta-Pi Networks: Preliminary Experiments

As discussed earlier in §2.1.3, the function defined by a DDD network is not continuous; thus a control system which uses a DDD network evaluator may be subject to Berliner's "blemish effect". This effect can occur whenever the controller has the option of choosing between actions which will lead it into different classes of the input domain. In my tic-tac-toe DDD network, the blemish effect was absent because on any given turn, all legal moves necessarily fell into the same class. In my backgammon DDD network, it is also *usually* the case that all legal moves will fall into the same pipcount class; however, in certain situations the system's choices do belong to different classes. An important example of such a situation occurs when the system must choose whether to break contact (turning the game into a straight race) or maintain contact. By the definition of my gating program, these two move choices will be evaluated by different subnetworks, so the blemish effect is theoretically possible here. In practice, the blemish effect did indeed appear in these positions, which MAESTRO frequently misplayed.[8]

The Meta-Pi modular network is immune from the blemish effect because it defines a function which is continuous over all regions of the domain. My method for training a Meta-Pi net was to freeze a trained DDD network, discard its hard-coded gating program, and then train the Meta-Pi gating network from scratch according to the equation given in §2.1.3. Using this method, I converted two DDD networks to Meta-Pi networks: the 0 hidden unit modular net (frozen after 800K training epochs), and the 20 hidden unit modular net (frozen after 171K epochs). For both of these, the Meta-Pi gating network had a topology consisting of 156 input units (the raw board position), 5 hidden units, and 12 output units corresponding to the 12 DDD subnetworks. The training data, as usual, was gotten by applying TD(0) learning to games generated by self-play.

Unfortunately, the Meta-Pi network was quite slow to generate training data because its output is composed from the results of not one but all 12 subnetwork evaluations. Figure 2-12 shows the learning curves for both Meta-Pi nets, and for reference shows as horizontal lines the performance of the frozen DDD networks which they were trying to exceed. In head-to-head competition, the trained Meta-Pi networks played to a statistical draw against the DDD networks which spawned them (see table 2.2). This shows that after relatively few training epochs, the Meta-Pi gating network has learned enough to be as effective as the hard-coded DDD gating program. With further training, it seems likely that the performance of the Meta-Pi network could continue to rise.

[8]For example, if MAESTRO is even in the race but has a stronger board than his opponent, he will waste pips in his inner board and refuse to break technical contact even if his opponent is extremely unlikely to have to leave a shot.

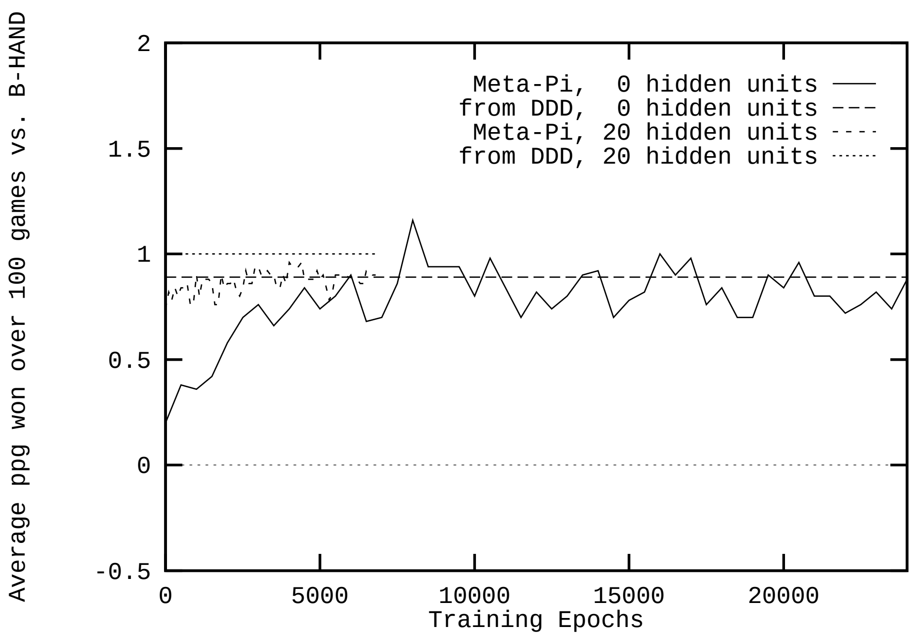

Figure 2-12: Performance of Meta-Pi Backgammon Networks

Chapter 3

Conclusions

In the three months allotted for this summer project, I have had time to investigate and synthesize some of the recent developments in reinforcement learning, game-playing, and modularity in neural networks. Although some of the graphs are sketchy and the training runs still incomplete, the experimental results I obtained do suggest several conclusions and directions for future research.

3.1 Summary

One of the main goals of this project was to replicate and confirm the results of [Tes92]—namely, that a connectionist network trained by the method of temporal differences is capable of learning a complex pattern-recognition and evaluation task. I demonstrated that such a network could learn strategies in a deterministic game, tic-tac-toe, by playing games against a teacher algorithm. The network thus trained soon far surpassed its teacher's playing ability. Furthermore, the tic-tac-toe network achieved its success without having to perform any lookahead search, which has been the crutch of most game-playing algorithms. Indeed, one characterisation of the TD training method is as a kind of dynamic programming algorithm which attempts to compile lookahead results into a static evaluation function.

In the domain of backgammon, a complex stochastic game, I confirmed Tesauro's findings that a TD-learning system could induce a significant amount of strategy solely on the basis of games generated by self-play. A 20 hidden unit monolithic MLP, whose inputs encoded only the raw board position and whose network parameters were not carefully optimised for quickest learning, raised its playing ability from a random move selector to a solid intermediate level within approximately one week of training on a DECstation 3100.

Inspired by Waibel's work with modular networks for phoneme recognition [WSS89, HW89], I tested two modular architectures in an attempt to enhance the ability of the game-playing network to change its strategic priorities depending on the current context. The first architecture, the DDD network, is simply a collection of monolithic subnetworks along with a hard-coded program to select an appropriate subnet for any given input. The DDD networks learned to play substantially better backgammon than the comparable monolithic nets, particularly in contexts where the game's usual heuristics needed to be violated. The second modular architecture, the Meta-Pi network, uses a separate gating network to weight the outputs of all the subnetworks. By analogy with Berliner's SNAC (Smoothness, Nonlinearity, Application Coefficients) methodology for evaluation functions, I hoped that the Meta-Pi network would be able to improve on the DDD net. However, perhaps because of the limited amount of training time, the Meta-Pi network was only able to equal the performance of the best DDD net, not surpass it.

3.2 Backgammon: Problems and Prospects

As my DDD network competed at the Computer Games Olympiad, one particularly serious flaw in its play became apparent: the network badly underestimated X's chances in positions where X had two pieces on his 10 or 11 point, and similarly overestimated X's chances when O had two pieces on her 10 or 11 point. This mis-evaluation was directly responsible for many of MAESTRO's weakest plays and worst doubling cube decisions. Later, I observed the same weakness in the 20 hidden unit monolithic network from which MAESTRO's subnetworks had been originally cloned.

The most likely explanation of this flaw is that the network evaluator weights those two inputs so negatively that, while training by self-play, it never actually tries the option of placing two pieces on those two points. Although I argued that backgammon's dice rolls will force the network into exploring many areas of the space, it does seem possible that if a feature is severely underrated (perhaps by virtue of an unusually low random initial weight on the shortcut connection from input to output), then the network may never try it, and thereby never learn anything about it.

If this explanation is correct, then such problems could be avoided in the future by simply biasing the random initialisation of the network so that all features were overrated. During self-play training, the feature weights would gradually settle down to their correct level, but no feature could be left untried. Another possibility would be to train multiple networks independently, then have them learn by playing games against each other. Playing against an opponent with a style different from its own could help a network explore new regions of the domain, and thereby improve its play.

One way to substantially improve the overall performance of the backgammon system would be to apply more human intelligence on the design end. As Tesauro has shown, the use of pre-computed input features causes a dramatic improvement in the quality of the evaluation function learned. Similarly, the DDD network's hard-coded gating program would undoubtedly be even more effective if it reflected a backgammon player's view of a useful partitioning of the space of backgammon positions. The Meta-Pi network could subsequently be trained to smooth the transitions between these classes of positions.

Ideally, we would like our network to be able to decide for itself during the learning process what topology provides it with the right amount of context-sensitivity. To this end, the Cascade Correlation architecture [FL90] seems promising because of its ability to dynamically build very high-order feature detectors. Such high-order feature detectors could provide an alternative to modular networks for achieving the goal of context-sensitive function evaluation.

3.3 Applications to Speech Processing

In this thesis, I have shown that the combination of temporal-difference training with connectionist networks can provide an effective solution to the problem of learning to predict in a complex, noisy domain. Can the model's success at backgammon pattern recognition be extended to speech pattern recognition? There are two important differences between the two problems, but I believe neither is an obstacle to the model's potential:

- Backgammon is Markovian, i.e. all the information needed to predict the eventual outcome is present in the input vector at every timestep. In speech, by contrast, the information relevant to the final outcome must be accumulated over time. This difference can be handled by the same techniques that have enabled neural networks to be trained by supervised-learning for phoneme recognition: recurrence and time-delay windowing. Recent research into the promise of TD learning for non-Markovian domains has been discussed in [LM92].
- The desired quantity at each step of a backgammon game is the continuous-valued probability of winning, whereas the desired output of a phoneme recognition problem is a classification decision among 50 or more discrete choices. Nevertheless, if the 50 output units are viewed as probabilities as in [RF91], I believe TD methods will be more effective than the supervised-learning method, which trains the network to output a single 1 and the rest 0's at each timestep. Such hard classification decisions can be inappropriate, especially during the first few speech frames of a phoneme. The TD methods will be better able to pinpoint the moment when a phoneme truly becomes distinguishable, and thus provide more accurate phone probability information to the HMM or other higher-level speech recognition system.

I conclude with a quote from Sutton, who is also optimistic about the application of TD learning to speech processing:

> Perceptual learning problems, such as vision or speech recognition, are classically treated as supervised learning, using a training set of isolated, correctly-classified input patterns. When humans hear or see things, on the other hand, they receive a stream of input over time and constantly update their hypotheses about what they are seeing or hearing. People are faced not with a single-step problem of unrelated

> pattern-class pairs, but rather with a series of related patterns, all providing information about the same classification. To disregard this structure seems improvident. [Sut88]

Acknowledgments

I would like to thank Frank Fallside, David Montgomery, Barney Pell and Tony Robinson for many helpful discussions about this project, and the Winston Churchill Foundation of the United States for funding my year of study in Britain.

Bibliography

[All88] L. V. Allis. A knowledge-based approach to connect-four. The game is solved: White wins. Master's thesis, Faculty of Mathematics and Computer Science, Free University, Amsterdam, 1988.

[All89] J. D. Allen. A note on the computer solution of Connect-Four. In D. N. L. Levy and D. F. Beal, editors, *Heuristic Programming in Artificial Intelligence: The First Computer Olympiad*, Chichester, England, 1989. Ellis Horwood Limited.

[BBB89] M. Burton, S. Baum, and S. Blumstein. Lexical effects on the phonetic categorization of speech: The role of acoustic structure. *Journal of Experimental Psychology: Human Perception and Performance*, 15, 1989.

[Ber79] H. Berliner. Backgammon computer program beats world champion. *Artificial Intelligence*, 14, 1979.

[Boy93] J. Boyan. MAESTRO 1.0: A modular neural network for learning context-dependent backgammon strategies by self-play. In D. N. L. Levy and D. F. Beal, editors, *Heuristic Programming in Artificial Intelligence 4: The Fourth Computer Olympiad*, Chichester, England, 1993. Ellis Horwood Limited. In press.

[CBK89] G. Collins, L. Birnbaum, and B. Krulwich. An adaptive model of decision-making in planning. In *11th International Joint Conference on Artificial Intelligence*, 1989.

[Dre81] M. Dresher. *The Mathematics of Games of Strategy: Theory and Applications*. Dover Publications, Inc., New York, 1981. First published in 1961 by Prentice-Hall, Inc.

[Fah88] S. Fahlman. An empirical study of learning speed in back-propagation networks. Technical Report CMU-CS-88-162, Carnegie Mellon University, 1988.

[FL90] S. Fahlman and C. Lebiere. The Cascade-Correlation learning architecture. In David Touretzky, editor, *Advances in Neural Information Processing Systems 2*. Morgan Kaufmann, 1990.

[FLW90] M. Franzini, K. Lee, and A. Waibel. Connectionist Viterbi training: A new hybrid method for continuous speech recognition. In *IEEE Proceedings of the ICASSP*, 1990.

[Hop82] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79, 1982.

[HW89] J. B. Hampshire and A. Waibel. The Meta-Pi network: Building distributed knowledge representations for robust pattern recognition. Technical Report CMU-CS-89-166, Carnegie Mellon University, August 1989.

[HWSS88] P. Haffner, A. Waibel, H. Sawai, and K. Shikano. Fast back-propagation learning methods for neural networks in speech. Technical Report TR-1-0058, ATR Interpreting Telephony Research Labs, November 1988.

[JJB91] R. Jacobs, M. Jordan, and A. Barto. Task decomposition through competition in a modular connectionist architecture: The what and where vision tasks. *Cognitive Science*, 13, 1991.

[Kan88] P. Kanerva. *Sparse Distributed Memory*. MIT Press, 1988.

[KBC88] T. Kohonen, G. Barna, and R. Chrisley. Statistical pattern recognition with neural networks: Benchmarking studies. In *IEEE Proceedings of the ICNN*, volume 1, July 1988.

[Koh88] T. Kohonen. The 'neural' phonetic typewriter. *IEEE Computer*, March 1988.

[LHR89] K. Lee, H. Hon, and R. Reddy. An overview of the SPHINX speech recognition system. *IEEE Transactions on Acoustics, Speech and Signal Processing*, January 1989.

[LM88] K.-F. Lee and S. Mahajan. A pattern classification approach to evaluation function learning. *Artificial Intelligence*, 36, 1988.

[LM92] L.-J. Lin and T. Mitchell. Memory approaches to reinforcement learning in non-Markovian domains. Technical Report CMU-CS-92-138, Carnegie Mellon University, May 1992.

[LN91] D. N. L. Levy and M. Newborn. *How Computers Play Chess*. Computer Science Press (W. H. Freeman and Company), 1991.

[Rab90] L. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. In Alex Waibel and Kai-Fu Lee, editors, *Readings in Speech Recognition*. Morgan Kaufmann, 1990.

[RF89] T. Robinson and F. Fallside. Dynamic reinforcement driven error propagation networks with application to game playing. In *Eleventh Annual Conference of the Cognitive Science Society*, 1989.

[RF91] T. Robinson and F. Fallside. A recurrent error propagation network speech recognition system. *Computer Speech and Language*, 5, 1991.

[RHW86] D. Rumelhart, G. Hinton, and R. Williams. Learning internal representations by error propagation. In D. Rumelhart and J. McClelland, editors, *Parallel Distributed Processing*, volume 1, chapter 8. MIT Press, 1986.

[RL81] L. Rabiner and S. Levinson. Isolated and connected word recognition—theory and selected applications. *The IEEE Transactions on Communications*, COM-29, May 1981.

[RMB+91] S. Renals, N. Morgan, H. Bourlard, M. Cohen, H. Franco, C. Wooters, and P. Kohn. Connectionist speech recognition: Status and prospects. Technical Report TR-91-070, University of California at Berkeley, December 1991.

[Rob92] B. Robertie. Carbon versus silicon: Matching wits with TD-Gammon. *Inside Backgammon*, 2(2), March-April 1992.

[Sam59] A. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3, 1959.

[SCT+92] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53(2-3), February 1992.

[Sut88] R. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3, 1988.

[Tes89] G. Tesauro. Neurogammon: A neural network backgammon learning program. In D. N. L. Levy and D. F. Beal, editors, *Heuristic Programming in Artificial Intelligence: The First Computer Olympiad*, Chichester, England, 1989. Ellis Horwood Limited.

[Tes92] G. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8(3/4), May 1992.

[Thr92] S. Thrun. Efficient exploration in reinforcement learning. Technical Report CMU-CS-92-102, Carnegie Mellon University, March 1992.

[Tou90] D. Touretzky, editor. *Advances in Neural Information Processing Systems 2*. Morgan Kaufmann, 1990.

[TS89] G. Tesauro and T. J. Sejnowski. A parallel network that learns to play backgammon. *Artificial Intelligence*, 39, 1989.

[Wat89] C. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, 1989.

[WSL89] A. Waibel, K. Shikano, and K. Lang. Phoneme recognition using Time-Delay Neural Networks. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 37, March 1989.

[WSS89] A. Waibel, H. Sawai, and K. Shikano. Modularity and scaling in large phonemic neural networks. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 37, December 1989.