TRAINING TECHNIQUES FOR SEQUENTIAL DECISION PROBLEMS

BY
CLIFFORD L. KOTNIK
B.S., INDIANA UNIVERSITY, 1976

A THESIS SUBMITTED TO
THE FACULTY OF THE GRADUATE FACULTY OF THE UNIVERSITY OF COLORADO AT COLORADO SPRINGS
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

2003
This thesis for the Master of Science degree by

Clifford L. Kotnik

has been approved for the department of Computer Science

by

_________________________________________
Dr. Jugal Kalita, Advisor

_________________________________________
Dr. Edward Chow, Committee Member

_________________________________________
Dr. Marijke Augusteijn, Committee Member

Date
CONTENTS

LIST OF ILLUSTRATIONS ......................................................... 6
LIST OF TABLES ................................................................. 7
CHAPTER I  INTRODUCTION .................................................... 8
CHAPTER II  RELATED RESEARCH ......................................... 11
      TD-Gammon ................................................................. 11
      Co-evolution and Self-play ........................................... 13
      Specific Evolutionary Algorithms ................................ 15
CHAPTER III  TEST PROBLEMS ............................................. 18
      Simplified Poker ......................................................... 18
      Gin Rummy ............................................................... 20
CHAPTER IV  IMPLEMENTATION ............................................. 23
      Common Agent State-Action-Policy Approach .................... 23
      Poker State-Action-Policy Logic ..................................... 25
      Gin Rummy State-Action-Policy Logic ............................... 26
      Artificial Neural Network .............................................. 28
      Algorithm Implementation ............................................. 29
      Runtime Architecture ................................................ 37
      Cluster Computing ...................................................... 38
CHAPTER V  ANALYSIS .......................................................... 40
      First Generation Gin Rummy Results ................................. 40
      Randomness and Bias .................................................. 43
      Second Generation Gin Rummy Results ............................. 45
LIST OF ILLUSTRATIONS

Figure 1 - Simplified Poker State Diagram .......................................................... 19
Figure 2 - Rummy Agent Model ............................................................................ 24
Figure 3 - Gin Rummy Look-ahead ...................................................................... 27
Figure 4 – Generation 1 TD-Rummy Turns per Game ........................................... 41
Figure 5 – Generation 1 Rummy Tournament Results ......................................... 42
Figure 6 – TD-Rummy Overtraining ..................................................................... 47
Figure 7 – Overall Generation 2 Rummy Tournament Results .............................. 50
Figure 8 – PSO-Poker Player Score versus Strategy .............................................. 58
LIST OF TABLES

Table 1 - Simplified Poker Equilibrium Strategy ........................................... 20
Table 2 - TD Player ......................................................................................... 30
Table 3 - Bout Pseudo-code ........................................................................... 31
Table 4 - PSO Player Pseudo-code ................................................................. 33
Table 5 - SANE Player Pseudo-code .............................................................. 34
Table 6 - Configurable Parameters ................................................................. 36
Table 7 - First Generation Rummy Training Parameters ............................... 41
Table 8 - Chi-squared Test of Generation 1 ................................................... 43
Table 9 - Shuffle Algorithms ........................................................................ 44
Table 10 - Effects of Bipolar Score for TD-Rummy Training ....................... 46
Table 11 - SANE-Rummy Algorithm Test Summary ..................................... 48
Table 12 - PSO-Rummy Algorithm Test Summary ....................................... 49
Table 13 - Typical Condor Job Timing ............................................................ 52
Table 14 - SANE-Poker Test Parameters ...................................................... 54
Table 15 - PSO-Poker Test Parameters ......................................................... 55
Table 16 - TD-Poker Test Parameters ............................................................ 56
Table 17 - PSO-Poker Learned Strategy ......................................................... 58
CHAPTER I
INTRODUCTION

The subject of machine learning addresses a number of different problems and encompasses a large number of techniques. One area, sequential decision problems, involve a number of decisions that are made as the state of the system changes. Examples include training robots to perform tasks, process control for dynamic systems, and playing games. Training for sequential decision problems is often complicated by the fact that correct behavior for the individual steps is unavailable. The correct behavior is only known at the conclusion of the whole sequence. However, the ability to address these challenges is often required for realistic problems.

Reinforcement learning has become a very successful area of machine learning that addresses sequential decision problems. Reinforcement learning can be defined (Sutton and Barto 1998, 4) as methods to solve problems characterized by delayed rewards and trial-and-error searches. Thus reinforcement learning is not one particular technique. The reward for a robot is obtained at the successful completion of a task. For a game, success is the score when the game is over. Games are often episodic, repeatedly having a series of steps followed by a reward. This need not be the case for reinforcement learning. Process control problems have rewards at regular intervals and are often not episodic. The analysis of reinforcement learning is usually framed (Mitchell 1997, 367) as an agent that knows or can sense the state of its environment and learns to
take actions based on experiencing the rewards of the various actions in these states. The strategy of action selection is referred to as the agent’s policy.

For some problems, the states and actions can be easily enumerated and held in tables. For many realistic problems, this is not possible. It is well known that game trees become extremely large for most non-trivial games. Even if large state-action matrices are stored, the time to train agents on all the combinations becomes computationally infeasible. Furthermore, for some problems, the state space is not static. What is needed are ways to generalize from training on specific states and actions to the general, possibly dynamic, problem.

Artificial neural networks (ANN) have been successfully applied in a number of situations. ANNs possess the capability to represent complex functions and to generalize from specific training. This approach is often called function approximation. Training ANNs for sequential decision problems requires different techniques from the more traditional classification problems, as there is no simple training set with target outputs.

TD-Gammon (Tesauro 1992) is an example of a very successful application of reinforcement learning with an ANN function approximation to the sequential decision task of playing the game of backgammon. Even more intriguing in TD-Gammon is the use of self-play for training. Rather than provide human expert moves to train the agent, the untrained TD-agent played against a copy of itself. Ultimately, the agent developed a level of play on par with human experts.

The ability of machine learning to, not only train agents on tasks involving complex sequences of actions, but to do so without specific human training is very compelling. This paper explores a number of aspects of the training techniques for
sequential decision problems, utilizing reinforcement learning and function approximation with ANNs.
CHAPTER II
RELATED RESEARCH

TD-Gammon

Tesauro trained an artificial neural network (ANN) to approximate the value function for the game of backgammon without explicit expert advice programmed into the agent. With only the definition of legal moves and a reward when the game was won, temporal difference (TD) learning and self-play allowed the ANN to be trained well into the level of experienced human play.

The ANN used for TD-Gammon was a conventional feed-forward neural network with one hidden layer and sigmoid activation functions. The inputs to the ANN encoded the number of pieces each player had on the 24 board positions, on the bar and pieces successfully removed from the board. The output of the ANN gave the probability of winning given that state. The number of hidden units varied from 40 to 80. For each roll, the player evaluated each possible legal move by applying the resulting state to the ANN and obtaining the probability of winning. The move with highest probability was selected. When the move resulted in the player winning, a reward signal of 1 was generated. Otherwise, the reward was 0.

Training was accomplished with temporal difference (TD) learning (Tesauro 1992). TD learning has more recently been formalized and expanded. See, for example, Sutton and Barto (1998). The major challenge in training the value function is that there are many steps the agent must take before the game is won and a reward can be assigned.
TD learning provides a method to assign credit from the reward to steps leading up to it. This is done in such a way that the value function can be adjusted in incremental steps as the game progresses. This approach provides an error signal that is back-propagated at each step of the game to incrementally train the network. The algorithm differs from normal backpropagation in that the history of weight changes over the course of the game is used at each step. Sutton and Barto refer to this history as the eligibility trace.

Tesauro (1992) reports on sensitivity of training to various parameters. The learning rate, \( \alpha \), has the usual meaning for ANN training. A value of 0.1 worked well. The TD algorithm includes the parameter \( \lambda \) that is used to decay the effect of a reward to actions in the past during training. \( \lambda \) takes on value in \([0,1]\). Tesauro found that training was not significantly impacted by the value of \( \lambda \), and used 0.7. The number of training games used ranged from hundreds of thousands in early experiments, to 1.5 million for later experiments.

One common technique used in TD learning (Sutton and Barto, 1998) is to force exploration of a wide variety of states by selecting a small number of actions at random, rather than always exploiting the optimal strategy. TD-Gammon required no such forced exploration. Apparently the stochastic nature of the roll of the dice forced enough different states to be visited.

Further refinements allowed TD- to reach expert level (Tesauro 1995). In fact, IBM, where Tesauro worked while developing TD-Gammon, consider the derived ANN weights to be confidential company property. This experiment is one of the best known successes for reinforcement learning and the TD training technique.
Co-evolution and Self-play

The self-play approach used in TD-Gammon is a very powerful feature. One study (Pollack, Blair and Land 1996) investigated how much of the success of TD-Gammon was due to the backpropagation TD technique. This group created a backgammon player using the same inputs, ANN, outputs and move selection policy. However, they adjusted the weights of the network using two competing agents and a simple hill-climbing, evolutionary approach. Since TD-Gammons weights are proprietary and not available to them, they tested against publicly available computer backgammon players with reasonable success. They conclude that reinforcement learning and TD training is not the primary reason for TD-Gammon’s success. They believe that the nature of backgammon lends itself to self-play, or co-evolutionary training techniques. However, if we take the definition of reinforcement learning, as above, to be methods to solve problems with delayed rewards and trail-and-error-search, then the authors’ approach seems to be a reinforcement learning approach, and it is the training technique they question.

Coevolution has become an area of study that most closely relates to the self-play aspect of TD-Gammon. In the more general sense, coevolution involves a population of agents that must be evaluated against each other. Angeline and Pollack (1993) define competitive fitness function to be any calculation of fitness that depends on the current members of the population of agents being trained. They point out that a single elimination tournament can be used to calculate the relative fitness, decreasing the computational requirements from $O(n^2)$ to $O(n)$.

In his PhD dissertation, Rosin (1997) studies the characteristics required for successful coevolution. The primary requirement is the notion of an arms race in which
closely matched opponents compete and improve together. The population must maintain diversity, or mediocre specialized agents will be produced that do not generalize well. Such agents have located local maximums in the search space. Rosin discusses the use of two populations: one designed to learn general solutions and another, the parasites, designed to defeat the general population in specific ways. The fitness of the general population can be evaluated in various ways. For example, in a technique called fitness sharing more weight may be given to agents that can defeat parasites that few other agents are able to defeat. Another technique, called hall of fame, keeps the best agents from earlier generations around, and current agents in the general population are evaluated against them.

Moriarty, Schultz, and Gerfenstette (1999) produced a very complete summary of the situation. They frame TD and evolutionary algorithms (EA) as two training techniques, both of which are applicable to reinforcement learning. They compare and contrast TD and EA. Both TD and EA are model-free, requiring no knowledge of the state transition rules. Both techniques address the credit assignment problem of applying the reward after a sequence of actions across those actions.

The paper notes a number of differences. TD systems normally represent the value function and derive the policy from it. This is because the normal TD training algorithm updates the value function. EA based systems can represent either the value function or the policy function directly. Where TD explicitly propagates credit back to the actions involved, EA implicitly assigns credit by proliferating agents which produce productive sequences of actions. TD implementations that are table-driven (as opposed to those with function approximation) can maintain knowledge of rare or poor states
better than EA implementations, since poor agents die out with EAs. Mutation and
crossover in EAs naturally provide a mix of exploration and exploitation, where TD often
requires exploration to be explicitly built in. The authors argue that EA methods are
more robust with hidden states or incomplete knowledge, but concede that both methods
suffer in this situation. Finally, the authors provide an argument that EA methods work
better when function approximation is used for generalization. They claim the less
frequent updates with EA make it less susceptible to having a single update alter the
global behavior for the worse. Moriarty, Schultz, and Gerfenstette (1999) conclude that
much work remains to fully understand the implications of these differences in practice.

Indeed, research is continuing. There are now gradient training techniques similar
to TD that operate on policy approximations (Sutton, et al. 2000).

**Specific Evolutionary Algorithms**

Many different evolutionary algorithms have been applied to training ANNs.
Most techniques are patterned after genetic algorithms where a population of agents
(individuals) are compared, based on a fitness function. The more fit agents reproduce to
make up the population in the next generation. Crossover and mutation provide the
diversity to explore the search space. The main difference when used in a coevolutionary
problem is that the fitness of the agents is measured by comparing the agents to each
other. In the case of a game, the agents can play against their siblings.

The symbiotic adaptive neuroevolution system, or SANE, is one evolutionary
algorithm that has been applied in a coevolutionary setting with success (Moriarty and
Miikkulainen 1997). With SANE, two populations are evolved by more or less
traditional genetic algorithmic techniques of crossover and mutation. SANE is applied to
ANNs with a single hidden layer. One population contains the single hidden units and all their associated connection weight, both input to hidden unit, and hidden unit to output. The second population of agents represents complete networks and are called blueprints. Thus, the blueprints are really just sets of individuals from the first population. The mutation and crossover operators function in a particular way to propagate hidden units and blueprints that perform well, while at the same time forcing new combinations to keep exploring the search space. The authors present results for SANE on a robotic simulation problem.

Another type of evolutionary algorithm, called cultural evolution, or particle swarm optimization (PSO) has been developed recently and applied to ANN training as well as other problems. This technique also contains a population of individuals that are ranked based on a fitness function. However, rather than simulating genetic reproduction, this algorithm simulates a cultural setting where individuals compare their performance, past and present, to their neighbors, and take on characteristics of the neighbor that is doing the best. Individuals track their own best performance along with their neighbor’s best, and change to a blend of both of these as well as a random component to maintain exploration. Thus the individuals in the population loosely follow the best individual in a pattern similar to a swarm of birds or fish. The rate at which individuals change to take on the characteristics of their neighbors is often decreased over the duration of the training run with a parameter called the inertial weight. This causes an effect similar to simulated annealing.

Kennedy and Eberhart (2001) describe the rational and technique in detail. They claim that they have used it to train ANNs faster than conventional techniques. They do
not discuss the use of PSO for coevolutionary problems. Van den Berg (2001) analyzes the theoretical basis for PSO and proves a number of results about the algorithm’s convergence. For situations where the objective function is non-stationary (as in coevolution of game players where the population learns new strategies) one technique often used is to periodically reset the particle’s memory so that its notion of its personal best is replaced by its current value, thus forgetting values that were derived based on an objective function that is no longer valid.

One interesting hybrid evolutionary approach has been described which combines SANE and PSO (Conradie, Miikkulainen, and Aldrich 2002). In this technique, called symbiotic memetic neuroevolution or SMNE, each generation uses an iteration of the SANE algorithm for broad searching, followed by an application of PSO for a focused local search. They claim good results on a controller for a simulated bioreactor.
Two test problems are investigated in this paper. Both problems involve training agents to play a card game using coevolution. The card games involve imperfect information or hidden states. They are two player games involving a sequence of alternating turns. Each game can be considered an episode with a score assigned to one of the players at its conclusion.

**Simplified Poker**

The first game is a simplified poker variant analyzed by Johannson and Niklasson (2001). They describe it as a variation on the game described by Kuhn (1950, 193-216). The deck of cards consists of one each of ace, king and queen. One player is designated the dealer and the other the gambler. There are at most three turns which consist of the actions fold, call or raise. These actions are taken by the gambler, dealer and finally the gambler again. Not all actions are permitted during each turn. Each player antes one dollar, and each bet or raise is for the same amount. The winner is the person with the highest card, provided his opponent calls. The winner takes all the money anted and bet. The money won is the reward in this reinforcement learning task. Figure 1 contains the state diagram for the game.

Johannsson and Niklasson analyze the game in terms of the equilibrium strategy and then conduct experiments to train agents to play the game using temporal TD learning and discrete state-action tables. They found that the agent’s strategy did not
converge to a steady state, but the average strategy did correspond to the equilibrium state. They then went on to show that if the agents played against opponents that did not use the equilibrium state, they could do better if they continued to employ TD learning while playing. Such online learning is a major advantage of the TD technique.

Figure 1 - Simplified Poker State Diagram

Of interest here is the theoretically derived equilibrium strategy. It provides an opportunity to test agents trained with various techniques against a known good strategy. The equilibrium strategy (Johansson and Niklasson 2001) is contained in table 1. Here the transition numbers for the actions refer to the states in figure 1. Experience has shown that this game is much more interesting than it appears at first glance. For
example, note the dealer’s strategy is to bluff with the queen and raise one out of five times.

<table>
<thead>
<tr>
<th>Hand</th>
<th>Turn→Action</th>
<th>Ace</th>
<th>King</th>
<th>Queen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gambler folds 2→7</td>
<td>0.00</td>
<td>0.00</td>
<td>0.67</td>
<td></td>
</tr>
<tr>
<td>Gambler bets 2→3</td>
<td>1.00</td>
<td>1.00</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>Dealer folds 3→5</td>
<td>0.00</td>
<td>0.67</td>
<td>0.80</td>
<td></td>
</tr>
<tr>
<td>Dealer calls 3→6</td>
<td>0.00</td>
<td>0.33</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>Dealer raises 3→4</td>
<td>1.00</td>
<td>0.00</td>
<td>0.20</td>
<td></td>
</tr>
<tr>
<td>Gambler folds 4→7</td>
<td>0.00</td>
<td>0.80</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>Gambler calls 4→6</td>
<td>1.00</td>
<td>0.20</td>
<td>0.00</td>
<td></td>
</tr>
</tbody>
</table>

Table 1 - Simplified Poker Equilibrium Strategy

**Gin Rummy**

The second test problem is to train an agent to play the game of gin rummy. Gin rummy is a two-handed card game that can be summarized as follows (Gibson 1974):

- **Deck:** standard 52 card deck.
- **Rank:** King=high, Ace=low.
- **Points:** King, Queen, Jack=10, Ace=1, all others=face value.
- **Deal:** 10 cards to each player; next card forms discard pile; remaining cards form the draw pile; discard pile is always face-up; draw pile is face-down; winner of each hand deals the next.
• Goal: form meld from sets of 3 to 4 cards of same value or sequences of 3 or more cards of same suit and with the total of the face value of remaining cards not so formed (called “deadwood”) less than or equal to 10; a single card cannot form part of a set and a sequence in the same hand.

• Turn: during each turn a player can take the top card from the discard or draw pile, must discard one card face-up on the top of the discard pile, and if the goal state is reached, may lay down meld and deadwood (called knocking).

• Play: players alternate turns starting with the dealer’s opponent until one player knocks.

• Laying off: after one player knocks, the opponent may extend any of the knocking player’s sets or sequences (called laying off) with any of his/her deadwood.

• Score: player who knocks scores the difference between the other player’s deadwood points and his/her own. If the player who knocks has no deadwood, the other player is not allowed to lay off, and the player knocking receives a score of 25 plus the other player’s deadwood points. If, after laying off, the opposing player’s deadwood points are equal to or less than the player knocking, the opponent scores 25 plus the difference in points instead of the player knocking.
Play usually continues until one player reaches 100 points. This portion of the game and other details of assigning bonus points beyond the description of a single hand are ignored for this analysis.

Gin rummy represents a moderately complex game that certainly cannot have all state-action combinations enumerated. Each card can be in the player’s hand, in the opponent’s hand, in the discard pile or in an unknown state of being either in the draw pile or the opponent’s hand. With 52 cards in one of four possible states there are $4^{52}$ possible states or approximately $2 \times 10^{14}$. On the other hand it has a simple set of rules and small set of actions at each turn.
An initial implementation of training for the gin rummy game (Kotnik and Kalita 2003) used TD backpropagation and a simple hill climbing coevolutionary algorithm similar to that described Pollack, Blair and Land (1996). The results of this experiment demonstrated that both TD and evolutionary approaches can be used to train agents to play gin rummy without explicit strategy being programmed in. Coevolution or self-play was sufficient to take agents from random play involving thousands of turns to games that completed in a couple dozen turns.

Encouraged by these results, a more complete experiment was conducted. A common training approach was developed for both the poker and gin rummy games. The training implements reinforcement learning using three techniques:

- temporal difference using backpropagation with eligibility traces (Sutton and Barto 1998)
- genetic evolution using the SANE algorithm (Moriarty and Miikkulainen 1997)
- cultural evolution using the PSO algorithm (Kennedy and Eberhart 2001) and (van den Bergh 2001)

**Common Agent State-Action-Policy Approach**

For all three training algorithms, a common approach for the agent’s states, actions and policy is used. There is a different state-action-policy approach for the two
different games with many common elements. Figure 2 shows a diagram of this approach for the gin rummy agents. The player and opponent are both implemented in the same way. As the players alternate turns, their roles in this diagram also alternate. The two players are simply different instances of the same agent with the same logic, but different ANNs.

Figure 2 - Rummy Agent Model

The state of the game is represented by the location of the cards which can be in the player’s hand, in the opponent’s hand, in the draw pile or in the discard pile. Each player has a different view of the state. Obviously, the role determines the sense of in-hand versus in-opponent-hand. Both players suffer from imperfect information. Each
Each player has its own ANN which produces a value based on a game state. The networks have a single layer of hidden units with sigmoid activation functions and a single output unit. Tests have been conducted with both sigmoid and identity functions for the output unit activation. The agent uses this state with a set of game specific logic representing its policy or strategy to select an action. If this action results in the game reaching its end, the game score is provided as a reward to the players. For both problems, games are always played in pairs with the role of dealer reversed. For gin rummy this means that the initial hand is identical, but the state in subsequent turns will vary given the differences between agents and the randomness of exploration, described below. The approach to pairing of games is due to Pollack, Blair and Land (1996).

**Poker State-Action-Policy Logic**

The only significant difference in the implementation of the two games involves the policy or strategy of how the agent selects the action based on the output of the ANN. For the poker agent, the ANN inputs represent current game turn as well as the current player’s card. The turn is encoded with three binary inputs representing the gambler’s first turn, the dealer’s turn and the gambler’s second turn. The output is the player’s value judgment of the hand at the current point in the game. The policy is implemented by generating a random number in the range [0..1] for each turn. For the gambler’s turns
that have two actions, if the ANN output exceeds the random number, the more
aggressive action is taken. For the dealer’s turn where there are three actions, the actions
are selected based on these cases:

- if ANN output > rand and ANN output > 0.5, then raise
- if ANN output < rand and ANN output < 0.5, then fold
- otherwise call

With this approach, the agent can create stochastic play where the ANN output controls
the frequency of selecting the possible actions.

The above logic is the greedy approach, representing the agent’s best efforts to
maximize the reward. With TD training, it is not always desirable to exploit the
maximum reward. Exploration is enabled at a variable rate during the algorithm by a
parameter in the range [0..1] representing the ratio of turns where the action is selected at
random from all possible actions in that state. This is known as the $\epsilon$-greedy approach
(Sutton and Barto 1998). This approach is implemented by first generating a random
number to decide between exploitation and exploration. If exploration is called for, a
second random number determines the action based on equal probability of legal actions
in the current state.

**Gin Rummy State-Action-Policy Logic**

The policy for the gin rummy agent employs a single level of look-ahead. Figure
3 shows how the look-ahead works. At the start of each turn, the player holds 10 cards
and can take the top card from the discard pile or the draw pile. The discard pile is
known, but the draw pile could provide any unknown card of which there can be at most
41. For each of these cards, the player must decide what it would discard if it had this
card in addition to its current hand. The game state resulting from discarding each of the 11 cards is run through the ANN and the choice is made that maximizes the ANN output. This is done for each of the possible cards that might be on top of the draw pile and the expected value of the draw pile is calculated. If the expected draw pile value exceeds the discard pile card, then the draw card is chosen. With the limited precision of floating point calculations and the nature of the sigmoid function, it can happen that multiple cards have the same value. When this happens, one of the equal options is chosen randomly. This proved particularly important for selecting the card to discard as a fixed order introduced bias.

Figure 3 - Gin Rummy Look-ahead
Two approaches were used to represent the inputs to the ANN. Both represent the state of each of the 52 cards as one of the following:

- in player’s hand
- unknown
- in discard pile
- in opponent’s hand

The smaller network was constructed by using a single input per card and assigning a value of 2 for in player’s hand, -2 for in opponent’s hand, -1 for in discard pile and 0 for unknown. Thus the input represented the strength of possession. A second approach used four inputs per card with a binary value representing one of the above conditions. This produced a network with four times as many weights.

For gin rummy, exploration was implemented in a way similar to poker. A random number first determined if the turn should be exploratory or greedy. Greedy turns used the logic described above. Exploratory turns first selected the pile to draw from randomly with equal probability. Then the card to discard was chosen at random.

**Artificial Neural Network**

The SNNS or Stuttgart Neural Network Simulator (Stuttgart 1998) publicly available software package was used for the core ANN functionality. The package consists of programs to design, analyze, train and run ANNs of many types. The features that proved useful for this experiment were the GUI program to define a network and the command line version of the network to propagate inputs to outputs. In addition, the
clearly documented data file definition format allowed the evolutionary algorithms to implement their custom training algorithms.

The formula used for backpropagation training of the ANN weights, $\Theta$, is

$$\Theta_{t+1} = \Theta_t + \alpha \left[ e_t + \gamma V'(s_{t+1}) - V'(s_t) \right] e_t$$

(1)

where $e_t$ is the vector of eligibility traces (Sutton and Barto 1998) that is built up over the course of each game based on the formula

$$e_t = \gamma \lambda e_{t-1} + \nabla_{\Theta_t} V'(s_t)$$

(2)

The factor $\gamma$ is a constant in the range $[0..1]$ called the discount rate. The term $\lambda$ is another $[0..1]$ constant that controls how strongly the eligibility trace applies to actions taken in the past. The immediate reward, $r_{t+1}$, is zero except when the player wins the game. In this case it is the score scaled to be in $[0..1]$. This specialize backpropagation is not implemented in SNNS. The command line program snns from the SNNS package was modified to implement this training algorithm for temporal difference learning.

For the SANE and PSO algorithms, the command line program was used during play to propagate the inputs through the network. Since the training for these population based algorithms is very different from any SNNS functionality, two separate programs were created to read the SNNS network files, alter the weights according to the algorithm, and rewrite the network file.

**Algorithm Implementation**

The algorithm for TD training is shown in table 2. Training occurs in regimes which consist of a large number of games for which the training parameters are held constant. Generally, parameters such as exploration frequency or learning rate are
gradually decreased over regimes in a manner similar to simulated annealing. Each player is represented by an SNNS network file that is initialized to random weights at the beginning of training. Within a regime, a small number of epochs are played, after which the networks are synchronized by arbitrarily copying one to the other.

```
1   td-player(training-parameters, champion, challenger)
2     for all regimes
3         set training parameters for this regime
4     initialize champion and challenger players
5     for all epochs
6         play-bout {champion, challenger,nbr-pairs}
7             copy champion network to challenger
8     end-for
9     end-for
```

Table 2 - TD Player

During an epoch, games are played in pairs. See table 3 for the pseudo-code. The deck is shuffled before each pair and the players are assigned to the roles of dealer and opponent. The opponent and dealer alternate turns until the game is over. After the first game of the pair, the deck is restored to the order obtained by the shuffle and the roles are reversed. Thus the players are each presented with the same initial hands. Due to random chance, the games are not played identically after that. Each turn consists of a random choice between exploiting the player’s assessment of the best action and exploring with a random move as described above. The rate of exploration is controlled by the program parameter $\epsilon$. To determine the best action, the policies for poker and gin rummy described previously, are used.
play-bout(champion, challenger, nbr-games)
loop nbr-games times
  shuffle the deck
  champion deals
  while game is not over
    take-epsilon-greedy-turn(challenger)
    update game state
    train the players
    if game is not over
      take-epsilon-greedy-turn(champion)
      update game state
      train the players
    end-if
  end-while
  record game statistics
  restore deck order to original shuffle
  challenger deals
  while game is not over
    take-epsilon-greedy-turn(champion)
    update game state
    train the players
    if game is not over
      take-epsilon-greedy-turn(challenger)
      update game state
      train the players
    end-if
  end-while
  record game statistics
end-loop

Table 3 - Bout Pseudo-code

If the turn results in the end of the game, the score for each player is calculated. For gin rummy, the score is calculated according to the rules of that game including the 25 bonus points for gin or an undercut. The score is assigned to the winner. In the case of poker, the amount bet by the loser is a positive score for the winner and a negative score for the loser. The score is scaled to the range [0..1] to form the reward. If the turn does not result in the end of the game, each player is assigned a reward of 0. This reward
is then used in the TD(λ) backpropagation algorithm to train each player’s network before the next turn. The results of each game are recorded in a statistics file for post processing.

The play-bout algorithm from table 3 is also used as the basic comparison of two networks to evaluate their relative merit with a few changes. In comparison mode, all turns are greedy, i.e. there is no exploration. In addition, no TD training takes place. Thus this same algorithm can be used to generate the fitness measurement in the evolutionary algorithms, and to test the results of one training run against another.

Two special purpose agents can also participate as players in play-bout. For gin rummy, a human player can provide the policy of what actions to take. A graphic display of the game state is presented and the human selects actions. Since the main purpose is to evaluate the strategy of the computer generated agents, both hands are open. For poker, a computer based player has been created which the equilibrium strategy hard-coded. The equilibrium player represents an expert opponent to test computer generated players in addition to playing them against each other.

The essential elements of the PSO algorithm are represented in table 4. In addition to various parameters, an ANN topology in the form of an SNNS network is input to the algorithm. A population of particles (specified by SNNS networks) is generated and updated during each generation. The inertia parameter is gradually decreased over succeeding generations to dampen the movement of the weights as they approach the optimal.

During each generation, a tournament is conducted. The tournament can be either a full bout of every player against every other player, or it can be a single elimination
tournament. The bout is conducted as outlined in table 3 without exploration or TD training. In the case of the former approach, the fitness of each player is their total score from all games. In the latter case, the fitness is the number of levels won. Once the fitness is determined, each particle’s best fitness value and associated weights are updated if the current tournament represents a new personal best.

```
1  pso-player(training-parameters, network-topology)
2  generate ANN for initial population of particles
3  initialize inertia
4  for all generations
5    conduct tournament to measure fitness of particles
6    update all particle personal best
7    for all particles
8      determine neighborhood or global best particle
9      update-weights(particlebest-particle,inertia)
10   end-for
11  decrease inertial-weight
12 end-for
13
14  update-weights(particle,best-particle,inertia)
15  for i := all particle-weights
16    velocity[i] := inertia * velocity[i]
17      + c1 * rand()
18        * (particle-best-weight[i] - particle-weight[i])
19        + c2 * rand()
20          * (best-particle-best-weight[i] - particle-weight[i])
21    clamp velocity[i] to min/max
22    particle-weight[i] := particle-weight[i] + velocity[i]
23  end-for

Table 4 - PSO Player Pseudo-code
```

The PSO weight update algorithm is then used to adjust the weights as shown on lines 14-19 of table 4. The best particle is the one with the best fitness of all particles in the swarm when the algorithm is operating in global mode. In local mode, the best is the maximum for the neighborhood of the current particle. The neighborhood is only those particles whose indexes happen to fall within a certain value of the current, where the size
of the neighborhood is a program parameter. These two approaches correspond to the
global and local PSO algorithms, respectively. The weights of the current particle are
adjusted by adding an amount, called the velocity, to the current weight. The velocity
changes with each generation. It consists of a random component, a component that will
move in the direction of the current particle’s best, and a component that carries it toward
the best particle’s best position. The velocity is also decreased by the inertia for each
generation. The constants c1 and c2 on line 16 are called acceleration constants.

1 sane-player(training-parameters, network-topology)
2 generate population of blueprints (ANNs)
3 generate population of hidden neural units
4 for all generations
5 conduct tournament to assign fitness of blueprints
6 assign fitness of blueprint to all hidden units contained
7 calculate average hidden unit fitness based on top blueprints
8 sort hidden units on fitness
9 loop nbr-hidden-units / 4 times
10 select 2 random parents from top 25% of hidden units
11 new hidden unit := crossover1(parent1, parent2)
12 new hidden unit := crossover2(parent1, parent2)
13 replace two hidden units in bottom 50% with new
14 end-loop
15 sort blueprints on fitness
16 loop nbr-blueprints / 4 times
17 select 2 random parents from top 25% of blueprints
18 new blueprint := crossover1(parent1, parent2)
19 new blueprint := crossover2(parent1, parent2)
20 replace two blueprints in bottom 50% with new
21 end-loop
22 mutate all hidden units
23 mutate blueprints in bottom 50% (i.e. offspring)
24 end-for

Table 5 - SANE Player Pseudo-code

Table 5 contains pseudo-code for the SANE evolutionary algorithm. The outer
loop is very similar to the PSO algorithm in terms of the generations and fitness
calculation. The weight adjustment is much more complicated. It is based on traditional
evolutionary operators of crossover and mutation. However, these are applied to two separate populations. The networks or blueprints are represented by SNNS network files. These networks are created from a subset of the other population, that of hidden units. In this implementation, all networks have a single hidden layer. Thus, a hidden unit is just the weights from each input to the hidden unit, the bias and the weight from the hidden unit to the single output. There is a many-to-many relationship between these two populations that changes from generation to generation. Each blueprint contains a number of hidden units. Each hidden unit is generally in a number of different blueprints.

When the tournament is conducted, a fitness value is assigned to each blueprint. The fitness of each hidden unit is calculated as the average of the best blueprints it participated in, where the maximum number of blueprints to average for a hidden unit is a configurable parameter. The two populations are ranked by fitness, and the bottom 50% of each population is replaced by the crossover offspring of random parents in the top 25%.

There are two crossover operations that operate essentially the same way for both blueprints and hidden units. The chromosome representing a blueprint is just a list of indexes of the component hidden units. The hidden unit chromosome is a list of weights. Both crossovers take two parents and produce one offspring. The first is a normal one point crossover. The second crossover simply selects one parent at random. The idea is that good parents should be preserved. By using direct copy of good parents, a more aggressive mutation rate can be used without detrimental effect.
The rate of mutation is controlled by program parameters. Mutation for hidden units is applied in the normal way to all members of that population after crossover.

Mutation for blueprints is applied only to the worst 50% of that population. It operates by first potentially switching each hidden unit randomly to a different one. Additionally, if the hidden unit is a parent in this generation, it may mutate randomly to one of its offspring.

While the poker game has a finite number of turns, gin rummy does not. The discard pile can be repeatedly shuffled and turned over to refill the draw pile. With initial random weights driving the gin rummy players, games can often take thousands of turns to complete. A draw limit was imposed, controlled by a configurable parameter. If the game exceeds this limit, it is considered a draw and neither player scores. If network changes have been made with TD backpropagation, the changes are discarded.

Table 6 summarizes the configurable parameters for the three algorithms.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Temporal Difference</th>
<th>Particle Swarm</th>
<th>SANE Evolutionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANN Inputs</td>
<td>52 or 208</td>
<td>52 or 208</td>
<td>52 or 208</td>
</tr>
<tr>
<td>ANN Output Activation Function</td>
<td>sigmoid or identity</td>
<td>sigmoid or identity</td>
<td>sigmoid or identity</td>
</tr>
<tr>
<td>ANN Weights</td>
<td>initial range</td>
<td>initial range and maximum velocity</td>
<td>initial range</td>
</tr>
<tr>
<td>Population size</td>
<td>always 2</td>
<td>nbr particles</td>
<td>nbr blueprints and hidden units</td>
</tr>
<tr>
<td>Training games</td>
<td>nbr regimes, epochs and game pairs</td>
<td>nbr generations</td>
<td>nbr generations</td>
</tr>
<tr>
<td>Algorithm specific</td>
<td>exploration rate, learning rate, $\lambda$, $\gamma$</td>
<td>inertia, acceleration constants</td>
<td>mutation rates</td>
</tr>
<tr>
<td>Gin rummy specific</td>
<td>draw limit</td>
<td>draw limit</td>
<td>draw limit</td>
</tr>
</tbody>
</table>

Table 6 – Configurable Parameters
Runtime Architecture

The training algorithms were implemented on Redhat Linux in the perl language. This choice may seem odd, as perl is not a terribly efficient language and is often thought of as a cryptic scripting tool. However, perl is a very powerful language and was chosen for speed of development. It also allows object oriented development and, with care, quite readable code can be developed.

Still the processing requirements of these algorithms are extremely large. Analysis showed that the ANN propagation and training were the most demanding from a processing standpoint. The SNNS package is written in C so it is much more efficient than perl. Therefore, the large body of new code to implement the algorithms and process statistics was done in perl. At the point the game state was generated, control passed to the SNNS module for processing. After profiling the resulting application, it was decided to move the gin rummy single step look-ahead logic into the C module with the ANN logic. In the resulting program, 87% of the CPU cycles are consumed in the C module when the state is represented as 52 inputs. With the larger ANN resulting from a 208 input state encoding, 96% of the CPU is consumed in the C module. Thus, the runtime impact of the less efficient perl language is minimized and the benefits for rapid development are still available.

For the gin rummy game, training cycles often ran for a number of days. To run tests trying a number of settings for the parameters required a great deal of computer time. The work was spread across a number of systems in the engineering lab. These systems often were rebooted for various reasons. The algorithms were implemented in such a way that intermediate results were written to disk at regular intervals, making restarts possible.
Cluster Computing

A number of projects have developed environments to take advantage of clusters of computers in various ways. One such environment is Condor (Condor 2003), developed at the University of Wisconsin-Madison. They describe their work as high-throughput computing. It is suited to tasks that can take advantage of coarse-grained parallelism on networked computers, either workstations or servers. It does not provide a high speed interconnect or specialized message passing facilities, although it complements some packages that do.

There are three main facilities provided. First, Condor provides job scheduling across a cluster of machines. Once a task is defined via a command language, it is submitted to a queue. Condor locates an appropriate machine, moves the required files to it, executes the job and returns the output files. The second component is the background processes that run on a machine providing shared resources to control the job execution and, if desired, balance the computing time between interactive human use and Condor jobs on the machine. For programs that can be linked with Condor, it provides automatic checkpoint and restart of jobs complete with remote snapshots of virtual memory.

The ability to recapture cycles on idle machines for compute intensive tasks like these training techniques is very appealing. To test the feasibility of applying cluster computer techniques to large training runs, Condor was setup and used for some training runs on this project.
In the PSO and SANE algorithms, the most compute intensive task is the tournament to calculate the fitness of a population. With a population of size $N$, a complete competition of all players against every other will require $N(N-1)/2$ matches. For a single elimination tournament $(N-1)$ matches are required. If a cluster of $N/2$ machines is available, the total time to process a single elimination tournament can approach $\log_2(N)$. In this test, two machines were available, so only a proof of concept was possible.

A tournament algorithm was created that split the tournament players in half, created the Condor command file, submitted it to Condor for execution and waited for the jobs to complete. Condor provides a simple perl interface to its job scheduling. Based on the command file, condor collected the required shell script and SNNS network files from the local machine, transferred them to a free Condor system, executed them and returned the statistical results. When both halves of the tournament were complete, the statistical results were analyzed to determine the player for the next round, and one final match determined the overall winner.
CHAPTER V
ANALYSIS

First Generation Gin Rummy Results

The results of the first generation of gin rummy algorithm (Kotnik and Kalita 2003) demonstrated the feasibility of self-play training for gin rummy and offered encouraging results from evolutionary techniques in addition to TD training. The TD approach used was very similar to that described in chapter IV, above.

The evolutionary technique was a simple hill-climbing technique described by Pollack, Blair and Land (1996) in their comparison to TD-Gammon (Tesuaro 1992). The population consists simply of two players. One is designated the champion, the other the challenger. Both start with random weights. After playing a bout in which the challenger wins, the champion’s weights are moved 5% toward the challenger. In either case, the challenger’s weights are mutated by adding Gaussian noise to them.

Both techniques succeeded in training players. Table 7 shows a summary of the players trained in the first generation tests. Upon examining the progress of training, the first characteristic that becomes obvious is the nearly unbounded nature of players starting from random weights. Games lasted for thousands of turns before one player would stumble across a winning hand. Initial results caused the reexamination of unit test results to see if the program contained a defect. However, as the number of games progressed, learning took place with both algorithms and the number of turns decreased.
<table>
<thead>
<tr>
<th>Name</th>
<th>Algorithm</th>
<th>Training Games</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD1</td>
<td>temp diff</td>
<td>9,484</td>
<td>alpha=0.1, lambda=0.3</td>
</tr>
<tr>
<td>TD2</td>
<td>temp diff</td>
<td>16,200</td>
<td>alpha=0.2, lambda=0.7</td>
</tr>
<tr>
<td>TD4</td>
<td>temp diff</td>
<td>16,243</td>
<td>alpha=0.2, lambda=0.2</td>
</tr>
<tr>
<td>TD5</td>
<td>temp diff</td>
<td>20,698</td>
<td>alpha=0.2, lambda=0.9</td>
</tr>
<tr>
<td>TD6</td>
<td>temp diff</td>
<td>1,800</td>
<td>alpha=0.2, lambda=0.9</td>
</tr>
<tr>
<td>EVO2</td>
<td>evolution</td>
<td>23,762</td>
<td>crossover=5%, mutation=0.1</td>
</tr>
<tr>
<td>EVO3</td>
<td>evolution</td>
<td>18,407</td>
<td>crossover=5%, mutation=0.1</td>
</tr>
<tr>
<td>EVO4</td>
<td>evolution</td>
<td>41,154</td>
<td>crossover=10%, mutation=0.1</td>
</tr>
</tbody>
</table>

Table 7 – First Generation Rummy Training Parameters

As shown in figure 4, after 1000 games, the average length of the training games decreased significantly, indicating learning taking place. The upturn in the line for EVO3 is the result of some extremely long games bringing the average up. This was prior to the
introduction of the draw limit to terminate a game after a certain number of turns. EVO3 did turn out to be the best player developed in the first generation. Figure 5 shows the results of tournaments conducted at various points in the training cycle of these players.

In examining the strategy of the players, a tendency appeared for the TD players to learn the simple approach of collecting a single suit. By doing so, a player will very likely get sequences and sometimes win. This strategy is a type of local optimal. Table 8 contains the analysis of the ending hands from a large number of games played in tournaments by the first generation players. The chi-squared test indicates the probability that the player would have ended up with the distribution of suits or face values if the these were chosen at random. It is expected that the face values would fail as low cards are preferred for knocking and minimizing the opponent’s score. However, there is no reason for bias in the suit.
Randomness and Bias

The initial test runs on the second generation algorithms started with the TD algorithm. Indications pointed to the players falling into the single suit strategy. The chi-squared test was applied to end game hands as well as the initial hands dealt. Two issues were uncovered.

The initial hands had some tendency to favor single suits. This was not always the same suit. Table 9, lines 1-11 show the original shuffle algorithm. This selects an element at random from the input array and adds it to the end of the output array. The random number generator used is the standard perl rand function that relies on the C library function of the same name. Since the size of the input array decreases, the range of the random integer generated varies, which is useful in minimizing non-random behavior. The input to this shuffle was the deck of cards, in order by suit and face value.

To increase the randomness of the shuffle it was found that a second shuffle helped, much in the same way a human dealer shuffles multiple times. In addition, a different algorithm was used for this second shuffle. Known as the Fisher-Yates shuffle, it is described by Knuth (1997, 145). Lines 12-20 show the code for the algorithm. This
also uses random integer generation across a variable range, but does an exchange to
shuffle in place.

```perl
sub random_permutation {
    my @in  = @_; 
    my @out = (); 
    my $i; 
    while ($#in >= 0){
        $i = int(rand($#in + 1)); 
        push (@out,splice(@in,$i,1)); 
    }
    return @out;
}

sub fisher_yates_shuffle {
    my $array = shift; 
    my $i;
    for ($i = @$array; --$i; ) {
        my $j = int rand ($i+1); 
        next if $i == $j;
        @$array[$i,$j] = @$array[$j,$i];
    }
}
```

Table 9 – Shuffle Algorithms

The second issue discovered was a bias in the gin rummy state-action policy code. In that algorithm, the future state generating the largest neural network output is selected. It was found that in a number of cases, the ANN output was generating a value for multiple future states that was identical within the precision of the floating point number. The original code selected the first such state. Since the states were evaluated in the order of the card index which, in turn, was based on suit and face value, this introduced a bias toward cards with lower indexes. The code was changed to randomly select one of the states with equal ANN output when multiples exist. The combination of these two changes made a significant improvement in training.
Second Generation Gin Rummy Results

In addition to the increased randomness of the shuffle and elimination of the bias, several other changes were made for the second generation:

• The game option of a player laying-off when his/her opponent knocks was implemented. The game played is now complete single hand gin rummy “according to Hoyle”.
• PSO and SANE algorithms were implemented to replace the simple evolutionary hill-climbing.
• The Condor proof of concept tournament was created.
• Game state representation as 208 inputs was added in addition to 52.
• Optional explicit exploration was added to TD training.

Work continued with the TD algorithm using the 52-input network topology from the first generation. The basic TD parameters were kept at the same levels as the first generation. These were 0.1 for the learning rate, 1.0 for $\gamma$ and 0.7 for $\lambda$. Exploration was added with a decreasing rate over a series of regimes. A useful level was found to be decreasing exploration from 0.2 to 0 in five equal steps over 5 regimes of 12,000 games each.

The TD player was now trained to a point where it was as good as the best player from generation one, EVO3. Training durations were extended, but the player reached a certain point and then performed worse. Tests of the end-game hands showed this overtraining was not due to learning a single suit strategy.

A change in the reward strategy for the TD training made a significant improvement. At the conclusion of the game, the loser was given a negative reward equal to the score of the winner. Previously, the loser received a zero reward on the final
hand just like all the other non-terminal hands of the game. This made a significant improvement in the training speed. Table 10 contains the scores from a tournament of EVO3 from generation one against several generation two players. All players are 52 inputs, 26 hidden units and sigmoid activation function on the output. T112 was trained with the bipolar final score change. T131 and T143 were after this change. However, the overtraining pattern continued after the bipolar change.

<table>
<thead>
<tr>
<th>Name</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVO3</td>
<td>3.00</td>
</tr>
<tr>
<td>T112</td>
<td>2.60</td>
</tr>
<tr>
<td>T131</td>
<td>4.20</td>
</tr>
<tr>
<td>T143</td>
<td>5.50</td>
</tr>
</tbody>
</table>

Table 10 – Effects of Bipolar Score for TD-Rummy Training

Next, the ANN topology was addresses. For the TD algorithm, tests were conducted with the identity plus bias function for the output unit, more hidden units and the 208-input approach. The impact for a larger network was positive, both with 52 and with 208 inputs. The 208-input network was tried with 26, 52 and 104 hidden units. The later gave promising results, but training times were very long. This is not surprising since the network contains 16 times as many weights as a 52 by 26 network, causing forward and backward propagation to be 16 times as long. Work continued with 208 by 52 as the largest network due to run times. The identify activation function did not prove useful.

With the TD algorithm tuned, an extended test was run using a 208 by 52 network for 13 regimes of 12,000 games, for a total of 156,000 training games. This player showed improvement for a longer period than any previous player, but did, in the end, go into a period of decreasing performance. Figure 6 shows the player at the end at each
regime. For each player, the training turns per game are shown on the left axis of that figure. A tournament between these players was conducted and the total score for each player is shown on the right axis.

Figure 6 – TD-Rummy Overtraining

The SANE algorithm tests are shown in table 11. After test N021, the fitness was determined by a single elimination tournament to speed the training. For N021, a single game pair was played between each player. For the subsequent training, five game pairs were played. Thus N021 had 552 games per generation and the others had 230 or 310, depending on the number of blueprints. All sessions used a population of 1272 hidden neural units. The topology of the network was fixed at 52 by 26.
The results show a preference for the identity function network topology. The scores are from a tournament played between every SANE player, and the generation one best players as well. The results indicate the players trained are on par with the evolutionary generation one player.

Table 12 shows a summary of the training sessions for the PSO algorithm. The larger 16 particle sessions give generally better results, as did the longer training sessions. For the number of generations run, overtraining did not seem to be an issue.

The acceleration constant was set to the value suggested by Van den Bergh (2001) for most sessions and varied for a few. Smaller initial weight ranges gave better performance and the velocity was kept at half the weight range. This keeps the resulting weight ranges from moving too far outside the initial weight range. Both the local and global best approaches were tried, with the global best giving slightly better results. A small range for the number of particles was tried. The type of activation function does not appear to be significant.
Table 12 – PSO-Rummy Algorithm Test Summary

One benefit that it was hoped for with the PSO algorithm was that the swarm of players would provide a range of strategies to compete against. The initial PSO results did not indicate a dramatic improvement in play against EVO3. To try to leverage the swarm in a different way, test S033 used a hybrid approach. Five particles were placed in the swarm and a full tournament with all players against all others was used for fitness calculation. However, during the games of the tournament TD training was turned on. Thus each generation contained two types of weight adjustments. This idea was inspired by Conradie and Miikkulainen (2001) who used a combination of SANE and PSO.

Based on the score, this was not very successful for S033.

One feature that the TD algorithm had after all the tuning was that it brought the player down to a reasonable number of turns per game more rapidly that SANE or PSO. Unfortunately, the TD algorithm still suffered from overtraining. To see if the PSO algorithm could take these partially trained TD players and improve upon them, a follow-up to the S033 session was run. First, eight identical TD training sessions were run on 208 by 52 networks all starting from different random weights. These were run for one
regime of 2000 games with exploration of 0.2. These were then placed into three swarms and run with the dual TD/PSO training described for S033, above. The resulting networks were actually a bit worse than the original single regime TD networks. See, for example, player X001 in figure 7.

Figure 7 shows a tournament between a number of significant players from the various tests. TD5 and EVO3 are the best players from generation one. T177 and T173 are the best and worst players from the single regime players trained for the combination TD/PSO test described above. The two T162 players are from the 156,000 game TD training session after the 9th and 12th regimes. The other players have been described above.

Figure 7 – Overall Generation 2 Rummy Tournament Results
Several things are noteworthy. First, there is a dramatic improvement from generation one. For this problem, TD has trained better players than evolutionary. This is reversed from generation one. Particle swarm is slightly better than SANE. Finally, the relative performance of the players can vary depending on the other players in the tournament. This is most pronounced for T143 that did not show much promise when compared to other 52 by 26 networks.

**Condor Cluster Tournaments**

In table 12, above, the first four agents listed used the PSO algorithm with the tournament executed in parallel across a two node Condor cluster. The two machines were both single processor Pentium 4 systems. One was 2.8 GHz, the other 2.4. The machines were connected with a 100BaseT Ethernet LAN. Each ran the Redhat Linux OS, version 8.0. Condor clustering software was at version 6.4.7. One of these two systems was also running the mainline of the PSO training algorithm.

Table 13 shows a typical Condor job near the end of the 800 generations of training session C002. Being near the end of the session, the games run faster due to training of the agents. Hence, the Condor overhead calculated will be near its highest for this session and the speed-up will be conservative.

In this session, there were 16 particles, so 8 were transferred to each Condor server. This involves transferring the 8 SNNS network files and associated shell script. The network files were 27 KB each for a total of 216 KB. This is shown in table 13 as “Run Bytes Received by Job”. Upon completion of the half tournament, a statistics file needs to be returned to the mainline for fitness calculation. This is the 60 KB shown as sent in table 13. Thus approximately 0.5 MB of network traffic is generated.
Table 13 – Typical Condor Job Timing

The Condor setup latency can be seen as the difference between the “Job submitted” and “Job executing” times. For this step the latency is 5-7 seconds. In addition to the network transfer, the Condor daemons on the execution machines must be contacted, checked for availability of resources, etc. This time is all overhead. One job ran for 208 seconds and the other for 207. The last line of table 13 shows the first line of job for the next generation. This shows that 18 seconds later the next condor job for the next generation is submitted. A rough estimate is that one third of this time is overhead
doing work that would not be done if Condor was not in used. That is six addition seconds of overhead on top of the seven seconds setup latency, for a total of 13 seconds of overhead.

So without Condor, this generation would have taken $207 + 208 + 12 = 427$ seconds. With condor, it took $7 + 208 + 18 = 233$. The speed-up is therefore 83% with two systems. Additional nodes in the cluster should be able to provide additional speed-up or allow larger swarms with the same duration. The key to diminishing returns will be when the unit of work distributed approaches the overhead. For the example above, the ratio of overhead to unit of work is approximately 16 times.

**Simplified Poker Results**

The poker player was developed after the TD, PSO and SANE training algorithms were developed for the gin rummy player. Surprisingly little change was required in the basic algorithms. The state-action policy was developed, some changes were required for the three-card deck and the statistical analysis was modified. After a few trial runs, it was decided to create a player with the fixed equilibrium strategy. This proved to be a useful way to obtain a fast, quantitative evaluation of performance.
<table>
<thead>
<tr>
<th>Test Name</th>
<th>Nbr Blueprint</th>
<th>Nbr Hidden Units</th>
<th>Nbr Gener</th>
<th>Weight Range (+/-)</th>
<th>Mu Offspring</th>
<th>Mu Weights</th>
<th>Mu Blueprt</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>n100</td>
<td>8</td>
<td>100</td>
<td>100</td>
<td>1.00</td>
<td>0.50</td>
<td>0.02</td>
<td>0.02</td>
<td>-630</td>
</tr>
<tr>
<td>n101</td>
<td>32</td>
<td>100</td>
<td>100</td>
<td>0.50</td>
<td>0.50</td>
<td>0.02</td>
<td>0.02</td>
<td>-239</td>
</tr>
<tr>
<td>n102</td>
<td>32</td>
<td>100</td>
<td>100</td>
<td>5.00</td>
<td>0.50</td>
<td>0.02</td>
<td>0.02</td>
<td>-159</td>
</tr>
<tr>
<td>n103</td>
<td>32</td>
<td>100</td>
<td>100</td>
<td>1.00</td>
<td>0.50</td>
<td>0.05</td>
<td>0.05</td>
<td>-343</td>
</tr>
<tr>
<td>n104</td>
<td>32</td>
<td>100</td>
<td>100</td>
<td>1.00</td>
<td>0.50</td>
<td>0.02</td>
<td>0.02</td>
<td>-274</td>
</tr>
<tr>
<td>n105</td>
<td>8</td>
<td>100</td>
<td>100</td>
<td>1.00</td>
<td>0.50</td>
<td>0.05</td>
<td>0.05</td>
<td>-311</td>
</tr>
<tr>
<td>n106</td>
<td>32</td>
<td>200</td>
<td>200</td>
<td>1.00</td>
<td>0.50</td>
<td>0.02</td>
<td>0.02</td>
<td>-144</td>
</tr>
<tr>
<td>n107</td>
<td>8</td>
<td>200</td>
<td>200</td>
<td>1.00</td>
<td>0.50</td>
<td>0.02</td>
<td>0.02</td>
<td>-181</td>
</tr>
<tr>
<td>n110</td>
<td>8</td>
<td>100</td>
<td>100</td>
<td>5.00</td>
<td>0.50</td>
<td>0.02</td>
<td>0.02</td>
<td>-62</td>
</tr>
<tr>
<td>n111</td>
<td>32</td>
<td>100</td>
<td>100</td>
<td>5.00</td>
<td>0.50</td>
<td>0.02</td>
<td>0.02</td>
<td>-172</td>
</tr>
<tr>
<td>n112</td>
<td>32</td>
<td>100</td>
<td>100</td>
<td>2.50</td>
<td>0.50</td>
<td>0.02</td>
<td>0.02</td>
<td>-351</td>
</tr>
<tr>
<td>n113</td>
<td>32</td>
<td>100</td>
<td>100</td>
<td>10.00</td>
<td>0.50</td>
<td>0.02</td>
<td>0.02</td>
<td>-403</td>
</tr>
<tr>
<td>n114</td>
<td>32</td>
<td>100</td>
<td>100</td>
<td>5.00</td>
<td>0.50</td>
<td>0.05</td>
<td>0.05</td>
<td>-1572</td>
</tr>
<tr>
<td>n115</td>
<td>8</td>
<td>100</td>
<td>100</td>
<td>5.00</td>
<td>0.50</td>
<td>0.05</td>
<td>0.05</td>
<td>-219</td>
</tr>
<tr>
<td>n116</td>
<td>32</td>
<td>200</td>
<td>200</td>
<td>5.00</td>
<td>0.50</td>
<td>0.02</td>
<td>0.02</td>
<td>18</td>
</tr>
<tr>
<td>n117</td>
<td>8</td>
<td>200</td>
<td>200</td>
<td>5.00</td>
<td>0.50</td>
<td>0.02</td>
<td>0.02</td>
<td>-1576</td>
</tr>
</tbody>
</table>

Table 14 – SANE-Poker Test Parameters

The smaller state and bounded duration of this poker game made it possible to exhaustively investigate a broad range of network topologies and algorithm parameters. For the ANN topology, tests were conducted with 6, 12, 18, 24 and 30 hidden units with both a sigmoid and identity function, for the output unit activation function. For SANE and PSO, single elimination tournaments were used in which 20 game pairs decided between each pair of players. Population sizes were in the range of 8 to 64. Thus for each generation of these algorithms there were 280 to 2520 games per generation. Tables 14 and 15 show the parameter settings tested for the population based PSO and SANE algorithms.
<table>
<thead>
<tr>
<th>Test Name</th>
<th>Nbr</th>
<th>Neigh</th>
<th>Nbr</th>
<th>Accel</th>
<th>Const</th>
<th>Weight Range (+/-)</th>
<th>Max Velocity</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>S100</td>
<td>8</td>
<td>8</td>
<td>100</td>
<td>1.5</td>
<td>1.0</td>
<td>0.5</td>
<td>-335</td>
<td></td>
</tr>
<tr>
<td>S101</td>
<td>16</td>
<td>16</td>
<td>100</td>
<td>1.5</td>
<td>1.0</td>
<td>0.5</td>
<td>-29</td>
<td></td>
</tr>
<tr>
<td>S102</td>
<td>32</td>
<td>32</td>
<td>100</td>
<td>1.5</td>
<td>1.0</td>
<td>0.5</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>S110</td>
<td>8</td>
<td>8</td>
<td>100</td>
<td>1.5</td>
<td>1.0</td>
<td>0.5</td>
<td>-140</td>
<td></td>
</tr>
<tr>
<td>S111</td>
<td>16</td>
<td>16</td>
<td>500</td>
<td>1.5</td>
<td>1.0</td>
<td>0.5</td>
<td>-374</td>
<td></td>
</tr>
<tr>
<td>S112</td>
<td>32</td>
<td>32</td>
<td>500</td>
<td>1.5</td>
<td>1.0</td>
<td>0.5</td>
<td>-118</td>
<td></td>
</tr>
<tr>
<td>S120</td>
<td>8</td>
<td>8</td>
<td>100</td>
<td>1.5</td>
<td>1.0</td>
<td>0.5</td>
<td>-78</td>
<td></td>
</tr>
<tr>
<td>S121</td>
<td>16</td>
<td>16</td>
<td>100</td>
<td>1.5</td>
<td>1.0</td>
<td>0.5</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>S122</td>
<td>16</td>
<td>16</td>
<td>100</td>
<td>1.5</td>
<td>1.0</td>
<td>0.5</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>S123</td>
<td>64</td>
<td>64</td>
<td>100</td>
<td>1.5</td>
<td>1.0</td>
<td>0.5</td>
<td>-54</td>
<td></td>
</tr>
<tr>
<td>S124</td>
<td>8</td>
<td>4</td>
<td>100</td>
<td>1.5</td>
<td>1.0</td>
<td>0.5</td>
<td>-33</td>
<td></td>
</tr>
<tr>
<td>S125</td>
<td>16</td>
<td>8</td>
<td>100</td>
<td>1.5</td>
<td>1.0</td>
<td>0.5</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>S126</td>
<td>32</td>
<td>16</td>
<td>100</td>
<td>1.5</td>
<td>1.0</td>
<td>0.5</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>S127</td>
<td>64</td>
<td>32</td>
<td>100</td>
<td>1.5</td>
<td>1.0</td>
<td>0.5</td>
<td>87</td>
<td></td>
</tr>
<tr>
<td>S130</td>
<td>8</td>
<td>8</td>
<td>100</td>
<td>1.5</td>
<td>5.0</td>
<td>2.5</td>
<td>-301</td>
<td></td>
</tr>
<tr>
<td>S131</td>
<td>16</td>
<td>16</td>
<td>100</td>
<td>1.5</td>
<td>5.0</td>
<td>2.5</td>
<td>-137</td>
<td></td>
</tr>
<tr>
<td>S132</td>
<td>32</td>
<td>32</td>
<td>100</td>
<td>1.5</td>
<td>5.0</td>
<td>2.5</td>
<td>-371</td>
<td></td>
</tr>
<tr>
<td>S133</td>
<td>64</td>
<td>64</td>
<td>100</td>
<td>1.5</td>
<td>5.0</td>
<td>2.5</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>S134</td>
<td>8</td>
<td>4</td>
<td>100</td>
<td>1.5</td>
<td>5.0</td>
<td>2.5</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>S135</td>
<td>16</td>
<td>8</td>
<td>100</td>
<td>1.5</td>
<td>5.0</td>
<td>2.5</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>S136</td>
<td>32</td>
<td>16</td>
<td>100</td>
<td>1.5</td>
<td>5.0</td>
<td>2.5</td>
<td>-72</td>
<td></td>
</tr>
<tr>
<td>S137</td>
<td>64</td>
<td>32</td>
<td>100</td>
<td>1.5</td>
<td>5.0</td>
<td>2.5</td>
<td>-56</td>
<td></td>
</tr>
</tbody>
</table>

Table 15 – PSO-Poker Test Parameters

For the TD algorithm training consisted of five regimes. Each regime was 100 epochs of 10 game pairs for a total of 10,000 games. Longer training tests did not result in any improvement. The TD training test parameters are shown in table 16. In the case of the exploration rate, the values were gradually decreased over the five regimes. For the $\lambda$ and $\gamma$ the values parameters, complete sessions of five regimes were conducted for each value.
After training, each network was tested in 1000 game pairs against the equilibrium player. The resulting score was generally negative, indicating an overall loss. Since each game involves one to three dollars, the maximum loss would be $6000.

One difference between the TD and evolutionary methods was the activation function. The TD algorithm worked much better with the sigmoid function, while SANE and PSO worked better with the identify function. The PSO algorithm worked better than SANE with the sigmoid function, and almost as well as TD. The TD and SANE algorithms worked their best with 6 or 12 hidden units. The PSO algorithm did reasonably well with all options for hidden units. The TD algorithm did better with very
small initial weights. Neither evolutionary algorithm seemed very sensitive to initial weights.

To compare the results, the network with 12 hidden units and the activation function performing the best for that algorithm was selected. The scores against the equilibrium player for this network topology is shown in the score column of tables 14, 15 and 16. For the TD algorithm, the values of $\gamma$ and $\lambda$ that worked the best were 0.6 and 0.7. The scores with these parameter values are shown in table 16.

It is clear from these results that the PSO algorithm worked the best. In nearly half the cases, it beat the equilibrium player, although by a slim margin. Given that the PSO algorithm also worked well with various ANN topologies, it appeared to be a very robust algorithm as well as the best performing. Excessively large numbers of particles are not required. Eight seems too small, but it did very well with 16 or 32. The algorithm did well with both local- and global-best approaches.

Another approach to analyzing the training is to examine the strategy the player learns. During the games played against the equilibrium player, the number of times each action is taken from each state is recorded and the ratio of actions calculated. The Euclidian distance between these values and those shown in table 1 are a measure of how similar the player’s strategy is to the equilibrium player. Figure 8 shows all the PSO players strategy during the test against the equilibrium player. The players have been sorted on score and their distance from the values of table 1 is also shown. Although the score reaches a break even point, the distance does not converge to the equilibrium
distance, indicating there are other ways to play that are almost as good as the equilibrium.

Figure 8 – PSO-Poker Player Score versus Strategy

<table>
<thead>
<tr>
<th>Turn→Action</th>
<th>Hand</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Gambler folds 2→7</td>
<td>Ace 0.00</td>
<td>0.00</td>
<td>0.46</td>
<td></td>
</tr>
<tr>
<td>Gambler bets 2→3</td>
<td>King 1.00</td>
<td>1.00</td>
<td>0.54</td>
<td></td>
</tr>
<tr>
<td>Dealer folds 3→5</td>
<td>Queen 0.00</td>
<td>0.65</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>Dealer calls 3→6</td>
<td>Ace 0.00</td>
<td>0.35</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>Dealer raises 3→4</td>
<td>King 1.00</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>Gambler folds 4→7</td>
<td>Queen 0.00</td>
<td>1.00</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>Gambler calls 4→6</td>
<td>King 1.00</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
</tr>
</tbody>
</table>

Table 17 – PSO-Poker Learned Strategy
Table 17 contains the strategy for a PSO player with a distance of 0.49 and score of -9, which is essentially a break even score. The table is in the same format at table 1. The strategy with an ace is identical to the equilibrium. With a king, the PSO player does not call the dealer’s raise with a king, while the equilibrium player does one out of five times. The PSO player has learned to bluff with a queen, being more aggressive with bluffing than the equilibrium player. However, this bluffing is limited to the gambler’s initial bet. The PSO player always folds with a queen as the dealer. These results are encouraging since even the discrete state algorithm used by Johansson and Niklasson (2001) did not converge to a steady state.
Self-play with reinforcement learning and an artificial neural network to approximate the value function for a game is an effective training method. This experiment has demonstrated that a number of techniques for adjusting the ANN weights can successfully be used. These techniques, when combined with self-play, can train agents on a set of rules without explicit human training.

The general approach is not without difficulties. The whole process is extremely compute intensive. With the population based evolutionary algorithms, it is possible to apply loosely coupled computer clustering techniques to speed-up the training time. In a simple prototype, a speed-up of over 80% was obtained with a two node cluster. Larger clusters and software to take advantage of idle workstations is now available and works well.

These techniques are heuristic techniques and require time to tune for the particular problem. There are a number of parameters the experimenter must set that need to be compared at various levels. This only makes the long training times more problematic. Self-play can lead the agents to be trained on a specific type of play that may not generalize, as with the single suit approach for TD. Bias or nonrandom behavior in the programs can lead to poor training as well.

The temporal difference technique, after major refinements from the first generation, performed the best on the gin rummy problem. It has the additional
advantage that it can be used online, in a continuous learning situation, adjusting to a changing environment.

The particle swarm technique shows real promise. It behaved predictably and proved the best technique for the poker problem. For that game it worked the best with a range of parameters and network topologies. Kennedy and Eberhart (2001, 378) describe using PSO to also adjust the slope of the sigmoid activation function. They claim this technique can be used to evolve the number of hidden units, as those with very small slope are effectively removed from the network. This would be an interesting addition to these training sessions as it would eliminate a certain amount of parameter tuning.

One area that is still required for the researcher to discover is how to frame the state-action policy in such a way that the ANN value function can be used to determine the actions. Other possible mappings exist for these problems. Experimentation with them would be an interesting follow-up.

The more elaborate fitness comparison techniques from coevolutionary research would make interesting additions. The hall of fame approach was used, manually, in the analysis to compare the different approaches. There is no reason this could not be automated. Other hybrid techniques may also be possible. For example, the TD training could easily be conducted with one player receiving TD($\lambda$) backpropagation while the other player is a fixed network from the hall of fame.

Both the SNNS and Condor packages were useful tools. They worked well and were clearly documented. It really helps to advance research when one is permitted to build on the work of others without reinventing software.
WORKS CITED


