Turian, Joseph

Automated Feature Selection to Maximize Learning in Artificial Intelligence

1 Introduction

Traditional artificial intelligence (AI) computer programs reach a plateau beyond which learning stops. Computer learning has not achieved its potential. The problem is that when using current computer learning models with current computational power, learning does not progress far enough. To improve computer learning methods, we look towards human learning. Humans do not consider pointillist art as mere dots. Rather, they mentally synthesize the points into discrete entities, the prominent features of the painting. By discarding irrelevant features and recognizing important ones, humans perform feature selection. Feature selection is the process of adding and removing features (salient characteristics) to the thought process so as to maximize intelligence and learning potential. Features are important in learning because they define how one views the world. Perhaps if computers were to able to perform feature selection, their processing would be more efficient and consequently they would learn more intelligently and more quickly. Feature selection as a learning method could thus maximize learning in artificial intelligence.

In previous research, algorithms that performed feature selection did not create new features. These algorithms used features specific to their respective learning environments and often required initial human interaction. For example, chess-learning programs using "feature selection" were pre-assigned analysis of pawn structure and center control, which had to be built-in by the programmer. Features were used, but the "feature selection" was performed by humans in the programming design stage. In [KOHA96], Kohavi and John describe the use of "wrappers" to find the optimal subset of a group of features. In [CARU94], [LIU95], [LIU196], [LIU296], [PFAH95], [PIRA95], and [VAFA93], the research also limits feature selection to feature removal. In [JOHN94], [KOHA94] and [KOHA95], the addition of pre-existing features is discussed, but new features are not created.

This paper describes two new techniques, **Randomly Generating Operations** (**RGO**) and **Incremental Addition** (**IA**), that actually create new features. **RGO** and **IA** are two algorithms that perform automated feature selection.

Automated feature selection, meaning the creation, addition, and removal of features without human intervention, is a new process that is introduced by this project. "Automated" indicates that no human interaction is required once the program is started and that there are no hand-crafted features. The agent, meaning a computer program that learns, is totally independent from humans and from its subject matter. The agent is independent from its subject matter because, for all the algorithm knows, it could be learning how to calculate lunar trajectories. **RGO**, **IA**, and automated feature selection in general are portable to any similar learning system because no tricks that are specific to the subject being studied[1] are employed.

Therefore, this project attempted to improve computer learning techniques and to augment an agent's decision-making strength by using this project's new methods of automated feature selection,[2] **RGO** and **IA**.

1.1 Why Learning?

Computers are good at tasks that humans are poor at (such as number crunching and total recall), but computers are poor at tasks that humans are good at (such as natural language comprehension and visual discernment). Conventional methods of making computers "human-like" have been ineffective. Traditional artificial intelligence computer programs using previous computer learning methods have not succeeded because the programs stop learning at a certain point.[3] This project shows that computer learning techniques improve significantly, however, when **RGO** is used in conjunction with neural networking and that computer learning techniques may improve when **IA** is used in conjunction with neural networking and genetic algorithms. Neural networks, which have been likened to the human brain, are a computer data structure used for approximating

[1] For example, a trick in chess would be using a library of openings.
[2] For the sake of brevity, feature selection will henceforth refer to my process of automated feature selection.
[3] See [RUSS95], pg. 139 and [TESA95], conclusion.

mathematical functions. Genetic algorithms implement computerized natural selection and mutation. Testing shows that the learning capacity of neural networks used in the base learning systems[4] is boosted by certain methods of feature selection. Learning seems to be the only way to humanize computers and this experiment could lay the groundwork for thinking machines.

1.2 Why Game-Playing?

Game-playing is the optimal field in which to implement a feature selecting artificial intelligence agent because games provide an accessible, discrete, and static domain. An accessible domain is one in which the agent has complete knowledge of the current state. A discrete domain in one in which the state has clear-cut values. A static domain is one in which the state will not change while the agent is deliberating. The beauty of games and accessible, discrete, static domains is that they focus the research on the hypothesis. For example, testing whether this project's method of feature selection works in a continuous (non-discrete) domain would throw an unwanted random element into the study. Game-playing algorithms are a paradigm for real-world algorithms, yet, unlike-real world algorithms, they have only one variable, the one being studied.

1.3 Why Backgammon?

Backgammon[5] was carefully selected from a number of diverse strategy games. Checkers and chess agents implement brute force[6] search techniques because of the relatively low branching factors in these games. Branching factor is the average number of moves possible in a given game position.[7] In checkers the branching factor is approximately 7, and in chess the branching factor is approximately 15. In backgammon, though, the branching factor is approximately 400. To do a 4-ply (2 move deep) search, the backgammon playing agent must

4 The base learning system is the learning agent whose intelligence one wishes to improve by adding feature selection.
5 See [DAMI95].
6 Brute force is an algorithm in which every possible value is searched to find problem solutions. In most real-world applications, brute force is grossly inefficient — it takes too long and requires too much computer power.
7 See [KRUS91], pp. 157-167.

examine 25.6 billion positions, while a chess playing agent needs only to look at 50 thousand.[8] Thus, backgammon seemed to be a game in which elegant, rather than brute-force, solutions were needed. Because of this necessity for elegance, backgammon was chosen as the domain in which to test feature selection. Backgammon, though it involves chance, is also governed by strategy. If many games are played, the probabilities are factored into the thinking and the player attempts to make the best move for the expected (average) dice roll.

1.4 TD-Gammon

Gerald Tesauro[9] implemented a backgammon playing agent, `TD-Gammon`, that used delayed reinforcement learning in neural networks. Delayed reinforcement learning is a learning technique in which the agent does not know if it has made the right decisions until the very end, when it knows whether it has won or lost the game. Tesauro, himself an accomplished backgammon player, added hand-crafted backgammon features to his agent. `TD-Gammon` is ranked within the top 20 players (humans and machines) in the world in backgammon and is the best backgammon playing AI agent.

Gregory Galperin, my mentor, attempted to duplicate Tesauro's results. In the spirit of pure computer learning (without human assistance), he removed the hand-crafted features implemented by Tesauro and used only two neural nets: one for the attack phase of the game and one for the race phase of the game. Galperin's agent with very few neural net units (one neural net unit is the digital equivalent of one human brain neuron) played at an advanced beginner level after several hours of training.[10] With many neural net units and after long training, Galperin's agent played at a level comparable to Tesauro's agent.

---

[8] (branching factor)^{ply} = positions. Thus, 15^4 = 50 thousand, 400^4 = 25.6 billion.
[9] See [TESA91] and [TESA95].
[10] Training means having the program learn by playing many games against itself.

In this project, Galperin's advanced beginner `TD-Gammon` program was used as the base learning system. I then wrote the feature-selection routines, and Gregory Galperin and I wrote the code to interface his `TD-Gammon` program with my feature-selection code.

1.5 Overall Goal

Backgammon agents' playing strength may be improved by adding or removing analysis of such features as number of blots, pip count, and defensive/offensive strength. Yet, defining features explicitly[11] is not true learning because it requires human intervention. This project's algorithms, **RGO** and **IA**, avoided this need for human intervention by creating the features themselves. The experiments attempted to automate maximization of playing strength through feature selection. To mimic true learning, the programs started with minimal domain knowledge[12] and ascertained the features by themselves. Thus, the programs were totally independent and self-sufficient.

The **RGO** and **IA** algorithms can be readily parallelized (implemented on multiple computers to augment computing power). Parallel versions of the algorithms are developed and recommendations for parallelization techniques are presented. These parallelization techniques are easily scaleable, as very little time in the algorithm is devoted to communication. Though some parallel algorithms place heavy emphasis on transmitting much data, this experiment's algorithms do very little data exchange and are mostly self-contained. Feature selection can be used with learning programs running on any parallel implementation, from a workstation cluster to a massively parallel super-computer.

---

[11] Explicitly defined features are hand-crafted features that were created by a human, not by a computer program.
[12] Minimal domain knowledge means knowing only the essentials. In backgammon, minimal domain knowledge is knowing just the rules.

2 Materials and Methods

In this section, the precise **RGO** and **IA** feature selection algorithms are described in full. Testing methods are explicated and scaleable methods of parallelization are given.

The pseudo-code for the base game-playing algorithm is shown in **Listing 2-1**.

Listing 2-1. Game Playing Algorithm.

```
While the game not done
   For each possible move
      Tentatively make the move
      Evaluate the position with the evaluation function
   Reloop
   Make the move with the best evaluation output
Reloop
```

The evaluation output is a real number between -3.0 and 3.0, inclusive, representing the strength of the position. An output of 0.0 indicates that both players are equally likely to win the game. A positive output indicates that the player evaluating the position is more likely to win the game. An output of 1.0 indicates that the player evaluating the position is definitely going to achieve a normal win. An output of 2.0 indicates that the player evaluating the position is definitely going to win a gammon. [13] An output of 3.0 indicates that the player evaluating the position is definitely going to win a backgammon. [14]

[13] A gammon is a difficult to achieve ending position that is worth twice as many points as a normal win.
[14] A backgammon is an extremely difficult to achieve ending position that is worth three times as many points as a normal win.

In backgammon, the board is defined by 211 Boolean values. Any board position can be represented by a 211-bit number. The evaluation function transforms this 211-bit input into a real number, the position evaluation output. The evaluation function, in programs playing non-trivial games like backgammon, is not totally accurate. Program training is improving the evaluation function. The more accurate a program's evaluation function, the better it plays. In this project, the evaluation function was a neural network. Neural networks, useful for approximating mathematical functions, are used to assess the positional strength.

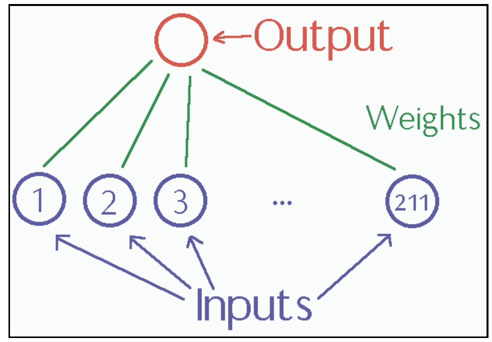

Figure 2-1. The base neural network.

The structure of the base neural net is shown in **Figure 2-1**. To calculate the strength of a certain position (the output, O), the program feeds-forward the board position, the 211 inputs. To feed-forward the board position, the program multiplies the inputs, I, by their respective weights, W, and sums the values. In formulaic notation,

$$O = I_1 \cdot W_1 + I_2 \cdot W_2 + \ldots + I_{211} \cdot W_{211}$$

Thus, to make a move, the program chooses the move that produces a position with the highest O.

At the start of training, the weights are randomly selected. To train the program, the weights are adjusted after each move is made according to Sutton's td(λ) formula:[15]

$$W_i[t+1] = W_i[t] + \alpha(O[t+1] - O[t]) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w O_k$$

This formula, unlike the the formula for O, takes the previous moves into account. $W_i[t]$ means the value of W_i, t moves into the game. α is a small constant, the learning gain. The lower the learning gain, the slower the program learns but the more able the program is to play well with excessive training. O[t] is the value of O, t moves into the game. $\nabla_w O_k$ is the gradient of O[t] with respect to each W. λ can be any value between 0 and 1, inclusive. If $\lambda = 1$, then the program sets the desired output to be the reinforcement given at the end of that game.

[15] See [SUTT88].

If $\lambda = 0$, then the formula can be reduced to:

$$W_i[t+1] = W_i[t] + \alpha(O[t+1] - O[t])\nabla_w O_t$$

At this extreme of $\lambda = 0$, the weight is incremented only by the most recent position evaluation. λ, as a real number between 0 and 1, smoothly interpolates between these two extremes.[16] In this project, λ is 0 during the training because, at this λ value, the program takes the least amount of time to train.

2.1 Feature Selection Methods

So far, traditional artificial intelligence algorithms have been described. The improvement upon these algorithms is feature selection. Because features were not explicitly defined, the program had to create the features itself. One of most important tasks in feature selection is operation addition. With more operations, a feature-selecting program will have more weights to adjust and thus the capacity for more learning. Operation addition is, according to this project's definition,[17] the combination of 2 inputs with a Boolean operator (AND, OR, etc.). So, if the program randomly added operation 212 = 106 OR 94, operation 213 = 205 AND 212, and operation 214 = 213 OR 53, the position strength approximation function would be

$$O = I_1 \cdot W_1 + I_2 \cdot W_2 + \ldots + I_{211} \cdot W_{211} + I_{212} \cdot W_{212} + I_{213} \cdot W_{213} + I_{214} \cdot W_{214}$$

See **Figure 2-2** (on the next page) for the graphical representation of the operation tree, the expanded form, of operation 214. The whole formula, expanded, would be:

$$O = I_1 \cdot W_1 + I_2 \cdot W_2 + \ldots + I_{211} \cdot W_{211} + (I_{106} \text{ OR } I_{94}) \cdot W_{212} + (I_{205} \text{ AND } (I_{106} \text{ OR } I_{94})) \cdot W_{213} + ((I_{205} \text{ AND } (I_{106} \text{ OR } I_{94})) \text{ OR } I_{53}) \cdot W_{214}$$

[16] See [TESA91], page 2.
[17] See **Section 3.3: Suggestions for Future Research** for proposals of different methods of operation addition.

The neural net needs no more input than the 211 observable features but has more weights to adjust to increase the playing strength. As an operation tree becomes huge and all-encompassing (by including many of the 211 observable features) from repeated operation addition, the operation tree itself becomes a feature, just as the composition of many molecules forms a cell. In chess, considering pawns as individual observables will not yield as great an insight as mentally combining the pawns to form the pawn skeleton. A chess program learns more if it first analyzes the pawns individually and then analyzes them collectively rather than if it just analyzes the pawns individually.

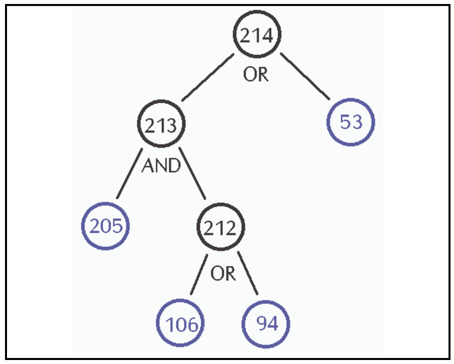

Figure 2-2. Operation Tree for Operation 214. The blue nodes[18] are observables. The black nodes are derived values. The tree represents the equation:

$$I_{214} = (I_{205} \text{ AND } (I_{106} \text{ OR } I_{94})) \text{ OR } I_{53}$$

The tree encompasses I_{53}, I_{94}, I_{106}, and I_{205}.

The methods of feature selection implemented in this project, **RGO** and **IA**, follow:

- **Randomly Generating Operations**- Randomly generating operations and testing them all at once
- **Incremental Addition**- Generating several new operations but adding only the one that increases performance the most

Other methods of feature selection were conceived but not implemented in this project. See **Section 3.3: Suggestions for Future Research** for proposals of different methods of feature selection.

2.1.1 Randomly Generating Operations

The pseudo-code for the **Randomly Generating Operations** algorithm follows in **Listing 2-2.**

Listing 2-2. Serial[19] Randomly Generating Operations Algorithm

```
Train the base program
Randomly create M operations (at the same time)
Train the new program
Have the new program play against the base program
Calculate the improvement
```

[18] A node is a datum in a tree, represented by a circle.
[19] Serial is the opposite of parallel. A serial algorithm runs on only one CPU, as opposed to a parallel algorithm.

The implementation, compiling, debugging, and testing of routines to generate operations randomly was performed. The source code is copyrighted (and thus unreleasable). The program was trained with 100 operations randomly added for N number of games and 1000 games to evaluate how much the feature-selecting program improved over the untrained program. N was varied throughout the testing.

2.1.2 Incremental Addition

Incremental Addition is a bit more subtle than **RGO**. **Incremental Addition** emulates natural selection. It starts with a base program, the parent, and spawns mutant offspring, each one with one new, random operation. These children develop to maturity (train). They then each fight with the parent (play backgammon) to find the child that improved the most over the parent. The most fit child (the best backgammon player) survives and becomes the new parent, while the rest are destroyed. The process is repeated many times.[20] The pseudo-code for **Incremental Addition** follows in **Listing 2-3**.

Listing 2-3. Serial Incremental Addition Algorithm

```
Train the base program (parent)
For every generation
   For each child
      Randomly generate one operation
      Train
      Play the new feature set against the base feature set
      Calculate how much the new feature set improved play
   Calculate which new feature set improved the most
   Replace the old feature set with the best new feature set
Reloop
```

The implementation, compiling, debugging, and testing of routines to add operations incrementally was performed. The source code is copyrighted (and thus unreleasable). Training was performed with the following parameters:

[20] The actual number of times the process is repeated depends upon what the programmer decides. In this experiment, the process was repeated 35 times.

35 generations of selection, 15 candidate children per generation, 1,000 games to train each child, and 250 games to calculate how much each child improved.

2.2 Parallelization

Performing training with many different feature sets can require a huge number of CPU cycles. Even though feature selection may require less computation than other techniques, real-world problems tackled with feature selection might still require more computing power than one CPU can supply. Fortunately, the algorithms are easily parallelized. A parallel algorithm is one in which many CPUs are used to augment computing power. The parallel algorithms presented are very effective because there is no parallelization overhead,[21] save the occasional millisecond of data transmittal. Of course, mere milliseconds are unimportant when speaking of a training period that can last anywhere between a few hours to a few weeks. Thus, because there is very little communication among the different CPUs, very little time is lost in data transmittal. Parallelization has been outlined, but not implemented, for **RGO** and **IA**.

2.2.1 Parallel Randomly Generating Operations

The pseudo-code for parallelization of **Randomly Generating Operations** follows in **Listing 2-4**.

Listing 2-4. Parallel Randomly Generating Operations Algorithm

```
Process 0[22] sends each processor the observable features
Each processor will then:
Randomly create operations
To play the game:
For each move
   Feed current features to linear net
   Send linear net output to Process 0
   Process 0 sums all the output
```

[21] Parallelization overhead is the extra computational time used to perform parallelization. For example, a program that takes 10 seconds on one CPU does not necessarily take 5 seconds on two CPUs. If, on two CPUs, the program takes 6 seconds, the parallelization overhead is 1 second (expected time - observed time).

[22] Process 0 is the program running on the initial computer. It controls and coordinates the parallelization.

```
   Process 0 makes a move
   Process 0 sends the new board to each process
   Each processor will then:
   Evaluate the move
   Send evaluation to Process 0
   Process 0 sums evaluations
   Process 0 sends it back to all others
   Each processor will then:
   Back-feed through the linear net
Reloop
```

2.2.2 Parallel Incremental Addition

The pseudo-code for parallelization of **Incremental Addition** follows in **Listing 2-5**.

Listing 2-5. Parallel Incremental Addition Algorithm

```
For every generation
   Process 0 sends the base feature set to each processor
   Each processor will then:
   Randomly generate one operation
   Train
   Play the new feature set against the base feature set
   Calculate how much the new feature set improved play
   Send improvement value to Process 0
   Process 0 calculates which new feature set improved the most
   Process 0 replaces the old feature set with the best new feature set
Reloop
```

2.3 Materials

During the summer, I performed research at the Massachusetts Institute of Technology Artificial Intelligence Laboratory in Cambridge, Massachusetts. Nearly all programming was done on **`boo-berry`**, an MIT AI Laboratory **`Sun-4/25`** that runs **`SunOS 4.1.4`**. Compiling was done on **`boo-berry`**, **`skydive`**, **`butterfinger`**, **`crystal-pier`**, **`taco-auctioneer`**, **`vector`**, and **`prophesy`**. **`skydive`** is Galperin's **`i686 PC`** that runs **`Linux.`** **`butterfinger`**, **`crystal-pier`**, **`taco-auctioneer`**, and

`vector` are AI Lab **`i686 PC`**s that run **`Linux.`** **`prophesy`** is my home **`i686 PC`** that runs **`Linux`**. Testing was done on various AI Lab **`Sun`**s, **`skydive`**, **`butterfinger`**, **`crystal-pier`**, **`taco-auctioneer`**, **`vector`**, and **`prophesy`**. At home in New York, I was able to telnet to the MIT AI Lab's systems, through the Internet. Over several months, I was able to continue to perform testing and further my research at home by using the MIT AI Lab's state-of-the-art computer systems.

3 Results and Discussion

3.1 Randomly Generating Operations Results and Data

Randomly Generating Operations has been tested the following way: the base (old) program was left untrained, 100 random features were added, the new program was trained for N games (N was set at different values), and the old program was played 1000 games against the new program to quantify how much the new program's playing strength has increased. To measure the increase in playing strength, a value called "equity" was created. Equity is a measure of how many games the new feature set won, on average, when playing against the old feature set. An equity of 1.0 means that the new feature set won a normal game against the old feature set in **every** game. An equity of 2.0 means that the new feature set won a gammon against the old feature set in every game. An equity of 3.0 means that the new feature set won a backgammon against the old feature set in every game. A **RGO** equity implies that the program has improved over the base (untrained) player if the equity is positive. Thus, equity is the new program's average improvement over the base player. The number of games played, N, was increased until the equities stopped changing, implying that a plateau had been reached. As a control, the program was run again, but no random features were added. The equities gathered for different Ns are shown in **Table 3-1**.

Table 3-1. RGO Output and RGO Control Output. For RGO (left side), N was stopped at 200,000 because the equity did not change very much between 150,000 and 200,000. For the control case (right side), N was stopped at 350,000 because the equity did not change very much between 300,000 and 350,000.

With **RGO** (feature selection)		Without **RGO** (no feature selection)	
N	equity	N	equity
50,000	-0.173000	50,000	-0.601000
100,000	0.488000	100,000	0.341000
150,000	0.949000	150,000	0.646000
200,000	0.958000	200,000	0.802000
		250,000	0.771000
		300,000	0.822000
		350,000	0.827000

The data in **Table 3-1** have been graphed in **Figure 3-1**. The solid red line is the graph of the equities with feature selection. The dotted blue line is the graph of the equities of the non-feature selecting control case. As shown in both **Table 3-1** and **Figure 3-1**, the equities with **RGO** are all higher that those without **RGO**. Because **RGO**'s equities are greater than the control case's equities, the program learned more with feature selection than without. In addition, with **RGO** the program plateaued after playing a fewer number of games. In terms of actual seconds, an i686/200MHz CPU can play 5.9 **RGO** games per second and 8.9 control games per second.[23] Regardless, RGO learns faster in shorter time. 200,000 **RGO** games take the same amount of time as 300,000 control games, but **RGO** learned more with 200,000 games than the control did with 300,000. Thus, **Randomly Generating Operations** is an effective method of feature selection because it improves a program's learning capacity and learning speed.

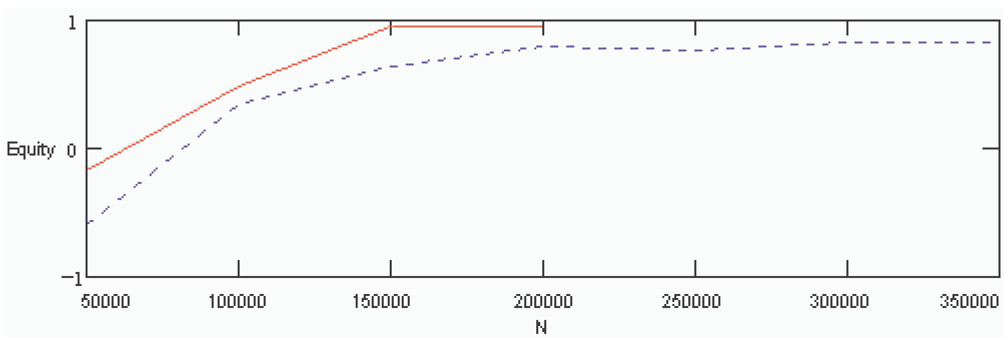

Figure 3-1. RGO Output and RGO Control Output. The solid red line is the output of the RGO with feature selection test case. The dotted blue line is the output of the control test case.

[23] **RGO** is expected to take longer time to train because the neural network has more weights to adjust.

3.2 Incremental Addition Results and Data

A demonstrative run of the **Incremental Addition** program yields the pertinent data shown in **Listing 3-1**. After each of the five children, NEW, is trained, the equities are calculated and the best child becomes the parent, OLD. As shown in **Listing 3-1**, every generation had at least one child that improved. Thus, the program constantly improved showing that evolution (fitness elevation), rather than devolution (fitness degradation), occurred. The maximum equities were rather high, 1.000000, 0.900000, and 0.800000, respectively. This testing shows how **IA** selects the most improved children.

Listing 3-1. Incremental Addition Output. Maximum equities are colored red.

```
Equity between NEW 0 and OLD: -0.100000
Equity between NEW 1 and OLD:  0.200000
Equity between NEW 2 and OLD: -0.600000
Equity between NEW 3 and OLD:  1.000000
Equity between NEW 4 and OLD: -0.800000
NEW 3.0 becomes OLD

Equity between NEW 0 and OLD: -0.700000
Equity between NEW 1 and OLD:  0.100000
Equity between NEW 2 and OLD:  0.200000
Equity between NEW 3 and OLD:  0.900000
Equity between NEW 4 and OLD:  0.300000
NEW 3.0 becomes OLD

Equity between NEW 0 and OLD:  0.800000
Equity between NEW 1 and OLD: -1.800000
Equity between NEW 2 and OLD: -0.200000
Equity between NEW 3 and OLD: -0.400000
Equity between NEW 4 and OLD:  0.400000
NEW 0.0 becomes OLD
```

The longest **IA** testing run had the following parameters: 35 generations of selection, 15 candidate children per generation, 1,000 games to train each child, and 250 games to calculate how much each child improved. Before performing feature selection, the program was trained 1,000 games. The maximum equities for the 35 generations are shown in **Table 3-2**. As a control, the program was run again, but no features were added. Thus, the parameters were as follows: 35 generations, 1 child per generation (no features added), 1,000 games to train the child, and 250 games to calculate how much the child improved. Again, the program was initially trained 1,000 games.

Unlike **RGO**'s equities, **Incremental Addition**'s equities represent an improvement over a constantly improving base program. In **RGO** testing, the base program's intelligence was fixed. A positive **Incremental Addition** equity means that the new program's playing strength increased, to some extent, in relation to the constantly improving base program.

Table 3-2. Incremental Addition Output. The maximum equities of Incremental Addition with 35 generations, 15 candidate children per generation, 1,000 training games, and 250 equity derivation games.

Generation	Equity	Generation	Equity	Generation	Equity
1	0.253000	13	0.233000	25	0.189000
2	0.080000	14	0.124000	26	0.201000
3	0.068000	15	0.281000	27	0.108000
4	0.072000	16	0.096000	28	0.137000
5	0.096000	17	0.353000	29	0.225000
6	0.112000	18	0.133000	30	0.133000
7	0.149000	19	0.173000	31	0.185000
8	0.044000	20	0.241000	32	0.120000
9	0.145000	21	0.273000	33	0.289000
10	0.104000	22	0.100000	34	0.229000
11	0.221000	23	0.084000	35	0.269000
12	0.221000	24	0.321000		

The equities for the control case are shown in **Table 3-3**. The negative equities, showing an intelligence decrease, are explained by initial adjustment and weight modification. This trait disappears after more training.

Table 3-3. Incremental Addition Control Output. The equities of Incremental Addition (control case) with 35 generations, 1 child per generation (no feature selection), 1,000 training games, and 250 equity derivation games.

Generation	Equity	Generation	Equity	Generation	Equity
1	-0.153000	13	-0.084000	25	0.032000
2	-0.177000	14	0.044000	26	0.012000
3	-0.213000	15	0.193000	27	0.008000
4	-0.076000	16	-0.317000	28	-0.072000
5	-0.072000	17	0.116000	29	0.028000
6	-0.088000	18	-0.181000	30	-0.068000
7	-0.205000	19	-0.016000	31	0.120000
8	-0.028000	20	-0.217000	32	-0.060000
9	0.032000	21	-0.189000	33	-0.309000
10	-0.016000	22	0.177000	34	-0.177000
11	0.189000	23	0.072000	35	-0.096000
12	0.044000	24	0.028000		

The data in **Table 3-2** and **Table 3-3** have been graphed in **Figure 3-2**. The solid red line is the graph of the maximum equities with feature selection. The dotted blue line is the graph of the maximum equities without feature selection, the control.

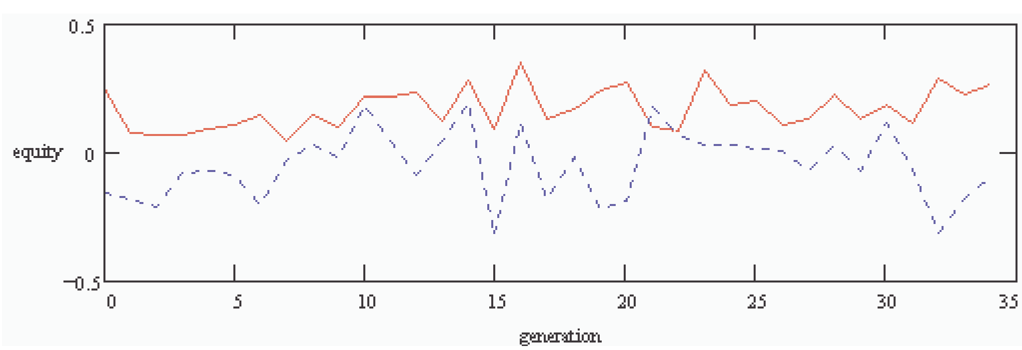

Figure 3-2. Incremental Addition Output and Incremental Addition Control Output. The solid red line is the output of the feature selection test case. The dotted blue line is the output of the control test case.

In the test runs shown in **Figure 3-2**, **Incremental Addition** was superior to learning without feature selection. Test runs of the **Incremental Addition** (see **Listing 3-1**, **Table 3-2**, **Table 3-3**, and **Figure 3-2**) indicate that automated feature selection in game-playing in artificial intelligence does improve an agent's decision-making strength. Results indicate that the program's intelligence has increased more than it would have without **Incremental Addition**. As of this writing, a test 100 times the magnitude of this test is being run to verify these initial results.

3.3 Feature Selection Discussion

Feature selection, as demonstrated by **RGO** and **IA**, can have great positive effects on the learning speed and learning capacity of an artificial intelligence agent. **Randomly Generating Operations** proved to be very beneficial. **Incremental Addition** methods are seemingly more principled than those of **RGO**, so more testing is underway to show under which circumstances **IA** will produce better results than **RGO**.

3.4 Suggestions for Future Research

There are many ways that this research can be expanded upon:[24]

- **IA** could be tested in comparison to **RGO** to determine which is a better method of feature selection. **RGO**'s results, in this paper, have been better than those of **IA**, even though **IA** better mimics human feature selection. Perhaps **IA** is better than **RGO** when extracting features from common human tasks, like speech recognition.
- Operation removal could be worked into the feature selection.
    - ♦ Searches greater than one level deep could be performed. Instead of adding one feature at a time, the program could perform several additions and removals at the same time. Redundant features could be pruned.

[24] Note that a two level deep suggestion assumes that the parent suggestion has been implemented.

- Feature selection could be tested upon other tasks besides backgammon. Many other tasks could effectively test the validity of and glean benefits from feature selection. Perhaps this project's techniques could be applied to the game Go, for which no computer has advanced past mediocre play. Several applications of other researchers' types of feature selection have been created. This project's feature selection algorithm could be trained with these publicly available data sets for comparative purposes. The data sets contain correct input and output for studies of audiology, breast cancer, chess, credit card applications, DNA, lenses, lung cancer, tic-tac-toe, and votes from the US House of Representatives. These data are used in [JOHN94], [KOHA94], [KOHA95], [LIU95], [LIU196], [LIU296], and [PFAH95].

- Parallelization routines could be implemented. The program's speed would increase by a factor of one hundred and testing would require less time.

- New methods of feature addition that use principle, as opposed to random creation, could be devised. The principle, on the other hand, would be independent of the specific subject. It would have to be applicable to other applications or else it would not be automated.

- Instead of having features defined as trees of Boolean operations, they could be defined as trees of polynomial operations. For example, instead of $I_{212} = I_{38}$ OR I_{162}, the operation could be $I_{212} = (I_{38})^2 + e^{(I_{162})}$.

  - ♦ Very interesting research could arise in this circumstance. For example, there might be a certain pattern of polynomial coefficients that maximizes equity. The equities could be graphed as a function of the coefficients and the program could attempt to extrapolate and interpolate equities with different coefficients. The program would then search to maximize the equity and use the coordinates as coefficients for the new parent.

- Multilayer neural networks[25] could be linearized. See **Figure 3-3** for an example of a multilayer neural network. A linear neural network is not sufficient for all problems. But training of multilayer neural networks is very time-consuming, requiring multi-dimensional error minimization (hill-climbing and gradient descent searches). Perhaps feature selection can implement linear neural networks on problems that typically require multilayer nets.

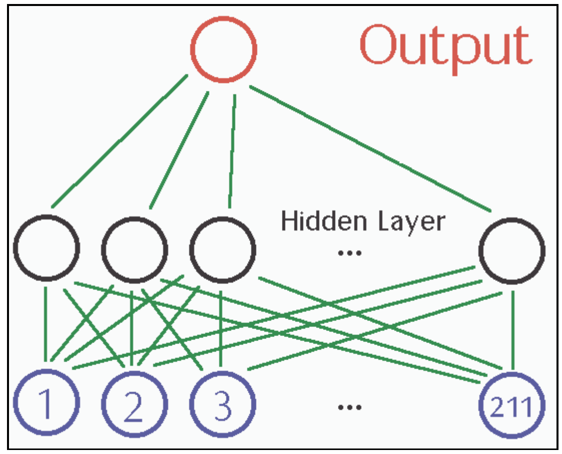

Figure 3-3. A Multilayer Neural Network.

- Operations could be added by performing infinitely recursive functions. This method would create "fractal" features.

- Features to create features could be created. For example, a certain feature to create features with a higher percentage of ORs could be created. Then, feature selection would be performed on the features to create features.

  - ♦ This concept could be carried out to greater degrees. Feature selection could be performed on features to create features to create features, infinitely recursed.

4 Summary

In this project, new learning techniques using feature selection were created and tested. Data obtained from experimentation and testing imply that an agent's decision making strength and intelligence can be substantially increased with automated feature selection. According to the results of full large-scale tests, **Randomly Generating Operations** greatly increases an agent's plateau strength (capacity to learn) and an agent's speed of learning (time taken to plateau). Promising, but still not fully plumbed, is **Incremental Addition**, which also

[25] A multilayer neural network is a neural network with at least one hidden layer. A hidden layer contains nodes that are neither inputs nor the output.

increased the agent's ability to learn. The algorithms developed are scaleable and easily parallelized with little time lost to transmission. More tests are being run and the future of feature-selection for training artificial intelligence agents looks bright.

5 Literature Cited

[CARU94] Caruana, Rich <*carauna@cs.cmu.edu*> and Dayne Freitag <*dayne@cs.cmu.edu*>. "Greedy Attribute Selection". *Proceedings of the Eleventh International Conference on Machine Learning*, pp. 28-36. http://www.cs.cmu.edu/afs/cs.cmu.edu/user/caruana/pub/papers/ml94.ps. 1994.

[DAMI95] Damish, Mark <*damish@ll.mit.edu*>. "Backgammon— Frequently Asked Questions". Last Modified: 15DEC95.

[JOHN94] John, George H. <*gjohn@CS.Stanford.EDU*>, Ron Kohavi <*ronnyk@CS.Stanford.EDU*>, and Karl Pfleger <*kpfleger@CS.Stanford.EDU*>. "Irrelevant Features and the Subset Selection Problem". In *Machine Learning: Proceedings of the Eleventh International Conference (ICML-94)*, pp. 121-129. ftp://starry.stanford.edu/pub/gjohn/papers/relevance4.ps. 1994.

[KOHA94] Kohavi, Ron <*ronnyk@CS.Stanford.EDU*> and Brian Frasca <*brianf@CS.Stanford. EDU*>. "Useful Feature Subsets and Rough Set Reduction". To appear in *The Third International Workshop on Rough Sets and Soft Computing (RSSC 94).* ftp://starry.stanford.edu/pub/ronnyk/rough.ps. 1994.

[KOHA95] Kohavi, Ron <*ronnyk@CS.Stanford.EDU*>. "The Power of Decision Tables". *The European Conference on Machine Learning (ECML-95), 1995*. ftp://starry.stanford.edu/pub/ronnyk/tables.ps. 1995.

[KOHA96] Kohavi, Ron <*ronnyk@sgi.com*> and George G. John <*gjohn@CS.Stanford.EDU*>. "Wrappers for Feature Selection". Technical Report, Computer Science Department, Stanford University. ftp://starry.stanford.edu/pub/ronnyk/wrappers.ps. 14AUG96.

[KRUS91] Kruse, Robert L. et al. *Data Structures and Program Design in C*. Prentice Hall, Engelwood Cliffs. 1991.

[LIU95] Liu, Huan <*liuh@iscs.nus.sg*> and Rudy Setiono <*rudys@iscs.nus.sg*>. "Chi2: Feature Selection and Discretization of Numeric Attributes". *Proceedings of the Seventh IEEE International Conference on Tools with Artificial Intelligence*. http://www.iscs.nus.sg/~liuh/tai95.ps. 1995.

[LIU196] Liu, Huan <*liuh@iscs.nus.sg*> and Rudy Setiono <*rudys@iscs.nus.sg*>. "Feature Selection and Classification — A Probabilistic Wrapper Approach". *Proceedings of the Ninth International Conference on Industrial and Engineering Applications of AI and ES*. http://www.iscs.nus.sg/~liuh/wrapper.ps. 1996.

[LIU296] Liu, Huan <*liuh@iscs.nus.sg*> and Rudy Setiono <*rudys@iscs.nus.sg*>. "A Probabilistic Approach to Feature Selection — A Filter Approach". *Thirteenth International Conference on Machine Learning (ICML '96)*. http://www.iscs.nus.sg/~liuh/ml96.ps. JUL96.

[PFAH95] Pfahringer, Bernhard <*bernhard@ai.univie.ac.at*>. "Compression-Based Feature Subset Selection". *Proceedings of the IJCAI-95 Workshop on Data Engineering for Inductive Learning*, pp. 109-119. http://ai.iit.nrc.ca/DEIL/pfahringer.ps.Z. 1995.

[PIRA95] Piramuthu, Selwyn. "Feature Selection and Backpropagation Neural Networks". *International Workshop on Neural Networks — Proceedings Volume I/II (1994/1995)*. http://www.leidenuniv.nl/medfac/fff/groepc/chaos/helnet/selwyn.htm. 1995.

[RUSS95] Russell, Stuart and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Engelwood Cliffs. 1995.

[SUTT88] Sutton, Richard S. "Learning to Predict by the Methods of Temporal Differences." *Machine Learning*, Vol. 3, pp. 9-44. ftp://ftp.cs.umass.edu/pub/anw/pub/sutton/sutton-88.ps.gz. 4FEB1988.

[TESA91] Tesauro, Gerald J. "Practical Issues in Temporal Difference Learning". *Machine Learning*, Vol. 8, No. 3-4, pp. 257-277. MAY92.

[TESA95] Tesauro, Gerald J. "Temporal Difference Learning and TD-Gammon". *Communications of the ACM*, Vol. 38, No. 3. http://www.research.ibm.com/massdist/tdl.html. MAR95.

[VAFA93] Vafaie, Haleh and Kenneth De Jong. "Robust Feature Selection Algorithms." *Proceedings of the 5th IEEE International Conference on Tools for Artificial Intelligence*, pp. 356-363. http://www.cs.gmu.edu/research/gag/papers/TAI93.ps. 1993.