An Empirical Study of Machine Learning Algorithms Applied to Modeling Player Behavior in a "First Person Shooter" Video Game

by

Benjamin Geisler

(Advisor: Professor Jude Shavlik)

A thesis submitted in partial fulfillment of

the requirements for the degree of

Master of Science

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

Fall 2002

Abstract. Modern video games present many challenging applications for artificial intelligence. Agents must not only appear intelligent but must also be fun to play against. In the video game genre of the first person shooter an agent must mimic all the behaviors of a human soldier in a combat situation. The standard opponent in a "first person shooter" uses a finite-state machine and a series of hand coded rules. Drawbacks of this system include a high level of predictability of opponents and a large amount of work manually programming each rule. Modern advances in machine learning have enabled agents to accurately learn rules from a set of examples. By sampling data from an expert player we use these machine learning algorithms in a first person shooter. With this system in place, the programmer has less work when hand coding the combat rules and the learned behaviors are often more unpredictable and life-like than any hard-wired finite state machine. This thesis explores several popular machine learning algorithms and shows how these algorithms can be applied to the game. The empirical study includes decision trees, Naïve Bayes classifiers, neural networks, and neural networks trained using boosting and bagging methods. We show that a subset of AI behaviors can be learned by player modeling using machine learning techniques. Under this system we have successfully been able to learn the combat behaviors of an expert player and apply them to an agent in a modified version of the video game *Soldier of Fortune 2*. The following tasks were learned: speed of acceleration, direction of movement, direction of facing, and jumping. We evaluate both empirically and aesthetically which learner performed the best and make recommendations for the future. We also have created a system which uses these learned behaviors in a finite-state system within the game at real time.

I. Introduction

Since their creation, the type of video games known as First Person Shooters (FPS) have been largely devoid of any sort of artificial intelligence. Games like Wolfenstein©, Doom© and Quake© present the player with enemies who mostly act as fodder. They walk towards the player, guns blazing, until they are dead. While this has its own merits, it has also become fairly boring. Recently developers began noticing this deficit and games such has Half-Life©, Counterstrike©, and UnReal© tournament have popped up. These games are successfully able to use expert-based systems and simple finite-state machines to give the illusion of an intelligent enemy (Linden 2001). As a result, these well established algorithms have helped game Artificial Intelligence (AI) advance in leaps and bounds in recent years. However, there is still much to be done. For the experienced player, the AI is never "good enough". It needs to keep challenging the player, to adapt to the player and if possible learn from the player. Currently there is no such behavior in the realm of the First Person Shooter. Instead, a good player will learn the behavior of the enemy AI and begin exploiting it. One of two things will happen at this point, either the player will abandon the game in boredom, or they will continue for entertainment value (after all, sometimes it's fun to let off steam). Clearly games must account for both types of players.

The answer to this conundrum may be found within the recent advances in academic artificial intelligence, especially machine learning. Machine learning is concerned with the question of how to build a computer program that can improve its performance on a task through experience. A task can be thought of as a function to be learned. It is the function's responsibility to accept input and produce output based on the input parameters. For example, in making a decision to move forward or backwards in a FPS the input to the function is a list of variables describing the current environment of the game. The input vector will include data describing how many opponents are near the player, the players health level, and any other relevant information. The output is a classification of whether or not the player should move forward. The decisions made by a human player in a first person shooter will often be unexplainable except by example. That is, we can easily determine input/output pairs, but we cannot easily determine a concise relationship between the input and desired output. We can easily collect data of a good player. We can sample his actions and based on his performance in the game we can state with confidence that these are examples of correct decisions or examples of incorrect decisions. This process is what we investigated in this thesis.

Machine Learning takes advantage of these examples. With these input/output pairs, a machine learning algorithm can adjust its internal structure to capture the relationships between the sample inputs and outputs. In this way the machine learner can produce a function which approximates the implicit relationships in the examples. Hidden amongst the examples will be many relationships and correlations. A human observer may be able to extract a few of these relationships but machine learning methods can be used to more extensively search out these relations.

To use a machine learning algorithm it is first necessary to determine the features that are important to the task. It is also necessary to determine what we will attempt to learn from our examples. Once these features are determined, we can represent an input/output pair as a collection of feature settings and an indication of the desired classification of this input. For example, the input/output pair in our move forward/backward function would be a vector of environment data about the game and a decision to move forward or backward as output. In our hypothetical task we would then collect samples of data that fit the “move forward” classification and samples of data which fit the “move backward” classification. The combination of these two types of example classifications makes up the data set. The learning algorithm makes use of both types of examples when changing its internal structure. This form of learning is known as *supervised learning*, since we are acting as a teacher to the learning algorithm (Mitchell 1997). We specify which examples should be thought of as positive and which are negative. A supervised learning algorithm will split our data set further into a training set and a test set. The training set serves to allow the machine learner to update its internal settings based on only these examples. The test set is used to estimate how well the internal learned function will perform on previously unseen data (Mitchell 1997).

There are many different machine learning algorithms to choose from. Some algorithms excel in certain domains while others do not perform as well. Instead of hastily picking a learning methodology it is preferable to frame the problem as much as possible. After the appropriate data set is determined it will be necessary to gather plenty of samples, and try a few standard learning algorithms. However, “standard algorithms” to academics are often novel ideas to game developers. For example, things like ID3 trees, neural networks, and genetic algorithms have only recently been considered. Machine learning is almost non-existent in the realm of the FPS video game. This thesis will show that progress can be achieved by applying some recent machine learning algorithms to the task of learning player behavior in a First Person Shooter. This will enable game developers to insert some unpredictability to the agents in a video game. It will be shown that this application will also allow developers to model expert players with little rule specifications. This means the doors will be opened for many different behaviors for our agents. For example, it will be very easy to model the behavior of a sniper or heavy-weapons specialist without needing predetermined rules.

With current trends in game development the timing couldn’t be better to introduce new learning algorithms. Computer gaming is now an six billion dollar a year industry (Savitz 2001). It is a competitive marketplace especially in terms of gameplay, which is something largely determined by the quality of a games’ AI routines. A game with a large variety of unique agent behaviors has a competitive advantage to a game with a few hand coded expert systems. In addition to being a preferred investment to developers it’s also a viable option on current hardware. Due to hardware advancements, processor cycles are easier to come by and memory is cheap. Neural networks in modern games would be unheard of in the late nineties. But now it’s not only possible on the PC, it’s already implemented in games such as “Black and White”™.

II. The Problem and the Data Set

It is straight forward to take a well-defined game, throw in some control structure for some agents, and begin teaching them. But an FPS[1] is not well defined. There has only been a spattering of attempts at doing so (Laird 2000). There are no defined tactics for a player of an FPS. The game is about battle, it is survival of the fittest; shoot or be shot. The more agents the player kills, the higher his score. But if the player is too gun happy, he might end up getting caught off guard and shot in the back. The penalties for death are severe; the player must wait thirty seconds before continuing the competition. Thirty seconds in this type of game is a significantly long time. In the meantime, the player's opponent might have picked up a better gun, or a "health" pickup, or finished off a mutual opponent whom he's been hunting since the beginning of the game (leaving the player with no points for the kill). Even expert players can't fully quantify their tactics. They can tell you general tips. i.e., make sure to not stay in one place too long, do not rush at an opponent straight on, strafe to the left if one can. But there are so many exceptions to any one rule that it makes it difficult to hard code a rule system.

To get a grip on all of this, the first step is to figure out what input the program has access to and what output might be useful. It might seem at first that an ideal FPS learning system would account for every action to take at any given situation. However, there will always be a need to allow some amount of programmer control in an FPS. The storyline and theme may demand specific actions at certain times. For example, it may be an interesting first encounter to set up a thug that always jumps out from behind a crate to scare the player. If we instead used player modeling to control the thug, the desired outcome may never happen. For this reason we want

[1] Although our research applies to any FPS, this research has used the FPS known as "Soldier of Fortune 2"™ (made by Raven Software) for its experiments.

control over what behaviors are learned from expert players and which are hand coded. Combat is a good place to use learned behavior because unpredicted (but sensible) actions are always preferred. Furthermore, there are only a handful of basic actions that are important during combat and these actions can be quantified. In this thesis we look at four of the basic actions: accelerate/decelerate, move forward/backward, face forward/backward, jump/don't jump. The general behavior of a bot[2] or player can be broken down into a sequence of moves (or move combinations). There will be many finer points to address once the sequence of moves is determined. For example, it's expensive to evaluate where a bot can move every time frame[3]. Without proper calculations a bot may accidentally bump into a wall or walk off a cliff. Such pragmatics will be addressed latter in this thesis.

What is needed to make a choice for any of these decisions? How does a player or agent know if it is time to move forward. This is at the heart of the problem. Typically these decisions are hand coded within some rule-based system. However, to achieve more lifelike and challenging characters, developers would like to model these decisions after an actual player and learn how these decisions are made. At this stage, the problem becomes a more traditional machine learning problem. A good set of features must be found to represent the environment of the game. This is the hard part of learning in a FPS, because even an expert gamer can not quantitize what it is that makes him good. Many of his decisions will be reflexive, and will not have an immediately tangible explanation behind them. With so much available data at any given time step, the selection of applicable features is very important.

[2] The AI agents in FPS video games are known as a "bots."
[3] A frame is the smallest time step in our simulation.

The Feature Set

The first step in modeling player behavior for an FPS video game is choosing the appropriate set of features for our learning algorithms. As mentioned, this thesis investigated four of the basic combat actions for an FPS: accelerate/decelerate, move direction, face direction and when to jump. These four actions will be the output portion of the data sample. We must now decide what information will be useful in making these four decisions. There is a large amount of available information but much of it is superfluous to learning player strategies. For example, knowing which 3D model file the player currently is using for his pistol is not useful to the learning algorithm. Perhaps the most important information in any combat scenario is to know where the agent's enemies are. The more knowledge our agent has, the better his chances at surviving. Since information about location is so important, it is divided into several smaller features. Spatial data about enemy location will be determined by breaking the world up into four sectors around the player, as illustrated in Figure 1.

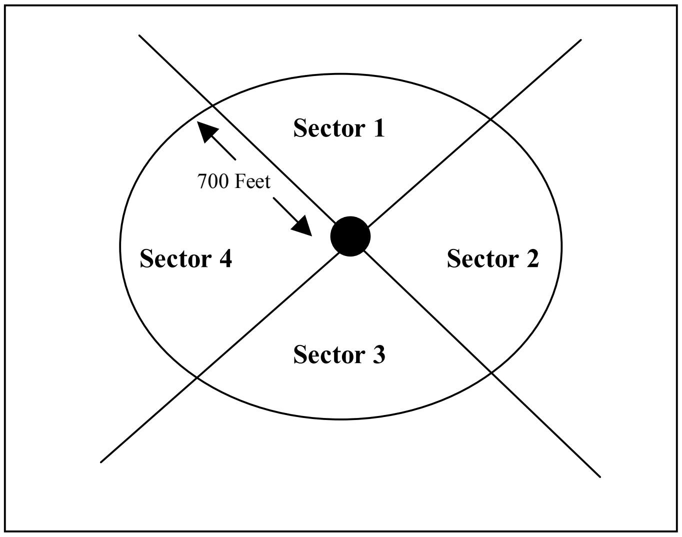

Figure 1: Game sectors with player centered in the middle.

Sector information is relative to the player. For example, Sector 1 represents the area in front of the player. Actual world position is not as important to the player as is information about his immediate surroundings. Data about enemies and goals will be expressed in terms of sector coordinates. For example, the location of the closest enemy to the player will be indicated by a sector number. If the goal is due east of the player it will be considered in sector 2. A sector ends at 700 feet and is the maximum view distance of the player on the sample levels.

Information about enemy location is extremely important to the player. Table 1 lists all the input and output features used for our learning algorithms. Of the twelve input features more than half of them report enemy statistics. The number of enemies in each sector will be recorded as will the health of the closest enemy, his distance from the player, and his sector location.

In addition to enemy information, the player must continually be aware of his health level. For example, if a player has low health his actions might be significantly different than if he has full health. Also, the Soldier of Fortune 2™ game is not simply about surviving the longest without dying. Points are awarded to whomever can "capture the flag" and return it to their base. To an expert player this is significant motivation to change strategy during game play. The feature set includes information about the distance the player is from the closest goal and what sector contains that goal. Finally, we record the current move direction and direction the player is facing. This allows us to determine such tactics as retreat and advance.

Table 1: Input Feature Set and Possible Outputs

Input (current information about the world around the player)	
Closest Enemy Health	This is the current health of the closest enemy. We discretize the possible values of this feature to zero through ten.
Number Enemies in Sector 1	This is number of enemies in sector one[4].
Number Enemies in Sector 2	This is number of enemies in sector two.
Number Enemies in Sector 3	This is number of enemies in sector three
Number Enemies in Sector 4	This is number of enemies in sector four.
Player Health	This is the player health. In the game the possible values are 0 to 100, we discretize this into 10 equal-sized bins.
Closest Goal Distance	Every game has a goal. By securing the goal the player gains points. The goal used in this research is a briefcase. Approximately 10,000 units is the start distance from the goal and this is the greatest distance possible; we linearly discretize this on a 0-10 scale.
Closest Goal Sector	The goal is in the following sector (note goal can be moved by other agent; this is non-static). Goal sector is relative to player.
Closest Enemy Sector	This is the sector containing the closest enemy.
Distance to Closest Enemy	This is the number of game units the agent is from his closest enemy. Values range from 0 through 10, scaled from a game representation of units (rounded to the nearest 1000 game units and capped at 10,000).
Current Move Direction	When this data was collected the player was moving in this direction (0 for moving south, 1 for moving north)
Current Face Direction	When this data was collected the player was facing this direction (0 for facing south, 1 for facing north).
Output (collected at next time step after input is sampled)	
Accelerate	If player is moving faster than in the last recorded frame, this variable is set to 1, otherwise the value is set to 0.
Move Direction	Direction of player movement in world angles; 0 means he is moving forward in the front 180 degree arc from world origin (north), 1 means he is moving backwards (south).
Facing Direction	This is the orientation of the player; 1 means he is facing somewhere in the front 180 degree arc of the world origin (north); 0 means he facing somewhere in the back 180 degree arc (south) .
Jumping	If the player jumped this frame, the value of this variable is 1, otherwise it is 0.

[4] If there are more than ten enemies the value is set to ten. The same upper bound holds for the other three sectors.

Extracting the Data Set

Once the feature set is determined it is possible to run the game and collect some samples. For the purposes of this thesis, the outcome associated with any set of features will be one of the basic actions as described in the output section of Table 1. The feature set will be measured every other game frame (100 milliseconds). Every sample has a corresponding outcome that occurs 50 milliseconds later (the amount of time it takes to process a combination of player key-press events). This outcome will be some combination of the four basic actions:

- Move front / back
- Face front / back
- Jump / do not jump
- Accelerate / do not accelerate

III. Learning From the Data

Data was collected by sampling the decisions of an expert player. The level used was a standard "capture the flag" game with thirteen enemy agents. We created a special version of Soldier of Fortune 2™ to collect the available game data and translate it into our feature vectors. For example, when a game frame is sampled we can access locations of the nearest enemies within a certain radius, then using vector math we compute their location relative to the player. This technique is used for all the sector information features. The *Move Direction* and *Face Direction* features are computed by simply recording the current world *Face Direction* and *Move Direction* of the player. The output section of the feature vector is computed on the time step following the sample. We record whether or not the player accelerates, changes movement, changes facing, or jumps. This part of the feature vector represents the decision made by the player. Now that we have the input features as well as the decision, we have a complete feature vector. This feature vector is saved and the collection of samples becomes our training and testing sets used for applying the learning algorithms. We collected over ten thousand individual examples by observing one expert player for 50 minutes.

A. Algorithm Descriptions

The first set of experiments will test a few standard classifiers: decision trees (ID3), Naïve Bayes, and artificial neural networks (ANN) to judge how well they can learn from our training examples.

ID3

There are many decision tree algorithms to choose from. This thesis uses the ID3 decision tree learning algorithm (Quinlan 1986). Since the decision at any point in time is not mutually exclusive of the other decisions, this problem can be easily extended to four separate decision trees. One decision tree will represent each possible binary decision: *Accelerate, Jump, Move Direction,* and *Face Direction.*

The training examples will be the feature vector at any time slice (the input), and the given action (the output). For each of the four decision trees the root node will be created and labeled with the feature that best classifies the examples. The standard information-gain formula was used. For the remaining examples, we recursively apply the ID3 algorithm and again find the best feature.

After several test runs, it was evident that ID3 generates some very lengthy decision trees. To overcome this it was decided to implement the common “rule post-pruning” algorithm. This algorithm uses a part of the training set as a *tune set* to estimate expected future accuracy. This accuracy information is then used to prune parts of the tree. In this thesis we pull out 10% of the training set to be used as a tune set for ID3 pruning. Rule-post-pruning can be summarized as follows:

1. Grow the decision tree as normal from train data.
2. Create one rule for each path from the root node to a leaf node.
   For example, assume we created this rule set:

   if (Enemy Sector = 1) ^ (Goal Sector = 2) ^ (Closest Goal = 2) then Jump = yes

   If (Enemy Health = 1) ^ (Enemy Sector = 2) then Jump = no
3. Calculate and save the tune set accuracy on the full set of rules.
4. For each precondition in each rule:
   a. Consider removing the precondition from the rule. For example, in the rule set above we would first consider removing (Enemy Sector = 1) from the first rule. On the next iteration we would consider removing (Goal Sector = 2). After considering the final precondition of the first rule, we move on to the first precondition of the second rule.
   b. Remove the precondition temporarily and score this new rule set using the tune data. Remember this score and the corresponding precondition removed.
5. Calculate the best tune set accuracy over all the rule sets created in step 4.
   a. If the best accuracy is not greater than the previous best accuracy, stop pruning and use the saved rule set from the *previous* step.
   b. Otherwise if the best accuracy is greater than the previous best accuracy, remove the corresponding precondition which, when absent, led to the higher accuracy. Save this rule set and return to step 4, considering this new rule set for further pruning.
6. Evaluate the final pruned rules by having them classify the test set.

NAÏVE BAYES

The Naive Bayes Classifier is another type of learner (Mitchell 1997). It uses Baye's theorem and assumes independence of feature values to estimate posterior probabilities. In this case it is anticipated that the features in and FPS are independent enough of each other to allow accurate classification. Classifications are made using the highest posterior probability. For example, if we have conditional independence then the probability of observing a conjunction of attributes Prob(**x** | c), where vector **x** contains the attributes and c is the result, is equal to the product of the probabilities for observing the individual attributes x_l. In other words Prob(**x** | c)= $\prod_{\{1,..,n\}}$ Prob(x_i | c) where *n* is the number of attributes in **x**. Thankfully our features are mostly independent of each other because most of the features concern location data. For example, the number of enemies in front of the player is not dependent on the number of enemies behind him. But there are certain instances in game play when this is not the case. For example, high player-health levels may dictate that more enemies are placed near the goal, meaning the enemy sector features may be partially dependent on the health feature. However, this is a rarity because this is only triggered when teams need to be evened out[5]. In addition, for many practical tasks Naïve Bayes still leads to accurate prediction even when the independence assumption is violated (Domingos 1996).

ANN

An artificial neural network (ANN) learns by using a training set to regress through the examples and learn in a non-linear manner. The basic back-propagation algorithm (with ten hidden units) was used in this project (Mitchell 1997).

[5] This phenomena only occurs approximately one in every twenty games.

The standard validation-set concept was used to avoid overfitting of the training set. Similar to the method of moving 10% of the training examples into a tuning set for ID3 pruning, 10% of the ANN training data will be moved into a tuning set to validate our learned function. After every five epochs of training we save the network weights at that time step and calculate the error on the validation set with these weights. If at any time the error rate is lower than the previous error rate from five epochs previous, training is stopped and the network weights of the previous validation step are used.

B. Experimental Methodology

Collecting the Data

Because of the complexity of the FPS there are some implementational hurdles to collecting this data. Since the data was sampled at a rate of every 100 milliseconds, there will be many input sets that look exactly the same. If nothing has changed in 500 milliseconds, only one of these samples is recorded. However, if at least one of the features changed this sample will always be recorded. This is important, otherwise crucial feature/action pairs might be missed.

Events with no corresponding action are discarded. A portion of these events may have proven useful: if no action is specified it could mean to stand still. However, rare events such as climbing ladders, opening doors, and going prone were not encoded. For this reason, it can not be assumed that the remaining non-classified situations dictate any particular action, so they were thrown out before learning began.[6]

[6] Un-classified examples accounted for approximately 7% of the total collected examples.

The game is run in multi-player mode with thirteen agents placed in a large multi-leveled environment. This environment includes places for every type of basic movement. This system for collecting data has been verified to work on any environment type with no necessary customization. However, the examples applied to the learning algorithms were always from the same environment. Data was collected over the course of several game sessions and combined into one massive data set of approximately 6000 examples. Each game was run by the same "expert" player, whose performance was fairly consistent.

Estimating the Error Rate of the Learned Models

After collecting the data each learning algorithm is applied in independently. After the algorithm has used the tuning set and training set, accuracy is estimated on the test set. The same test set of a thousand examples is used for all experiments. To get an idea of how much data will be needed in practice, each set of experiments was run on several different training set sizes.

C. Results

Figure 2 shows performance on the test set as a function of the size of the training set. The three basic algorithms exhibited satisfactory performance with error rates being low across all tasks. In general, the ANN always had lower error rates across all four tasks on the larger data sets sizes. On the smaller data sets, ID3 and Naïve Bayes sometimes achieved lower error rates than the artificial neural network (ANN). The lowest error rates for all four tasks were achieved by the ANN using a data set size of 5,000 examples. Figure 2 also shows a *baseline* error rate. The baseline illustrates the error rate achieved if the majority category is always guessed.

For *Face Direction* the ANN began to achieve lower error rates than the other two algorithms at only 500 data set examples. The error rate on *Face Direction* task dropped all the way to 16%. The baseline hovered at about 40%.

For *Move Direction* the ANN began to achieve lower error rates than the other two algorithms at 1,500 data set examples. The error rate on *Move Direction* dropped all the way to 7%. The baseline hovered at about 40%t.

For *Jump* the artificial neural network began to achieve lower error rates than the other two algorithms at only 500 data set examples. The error rate on *Jump* dropped all the way to 14%. The baseline error rate hovered at about 3%. The error rate baseline for this decision is actually substantially lower than the error rate using ANN. This interesting result is due to few occurrences of positive *Jump* examples. In relation to the other recorded decisions, jumping is a rarity. Using "confusion matrices" we will investigate this issue in full in another section. Because of the nature of jumping in an FPS, it turns out a higher error rate on this task is acceptable. A complete discussion is presented later in this thesis.

For *Accelerate* the artificial neural network began to achieve lower error rates than the other two algorithms at 1,500 data set examples. The ID3 error rate behavior was similar to the ANN error rate on this task. But the ANN was still more accurate; the error rate on this task dropped to 16%.

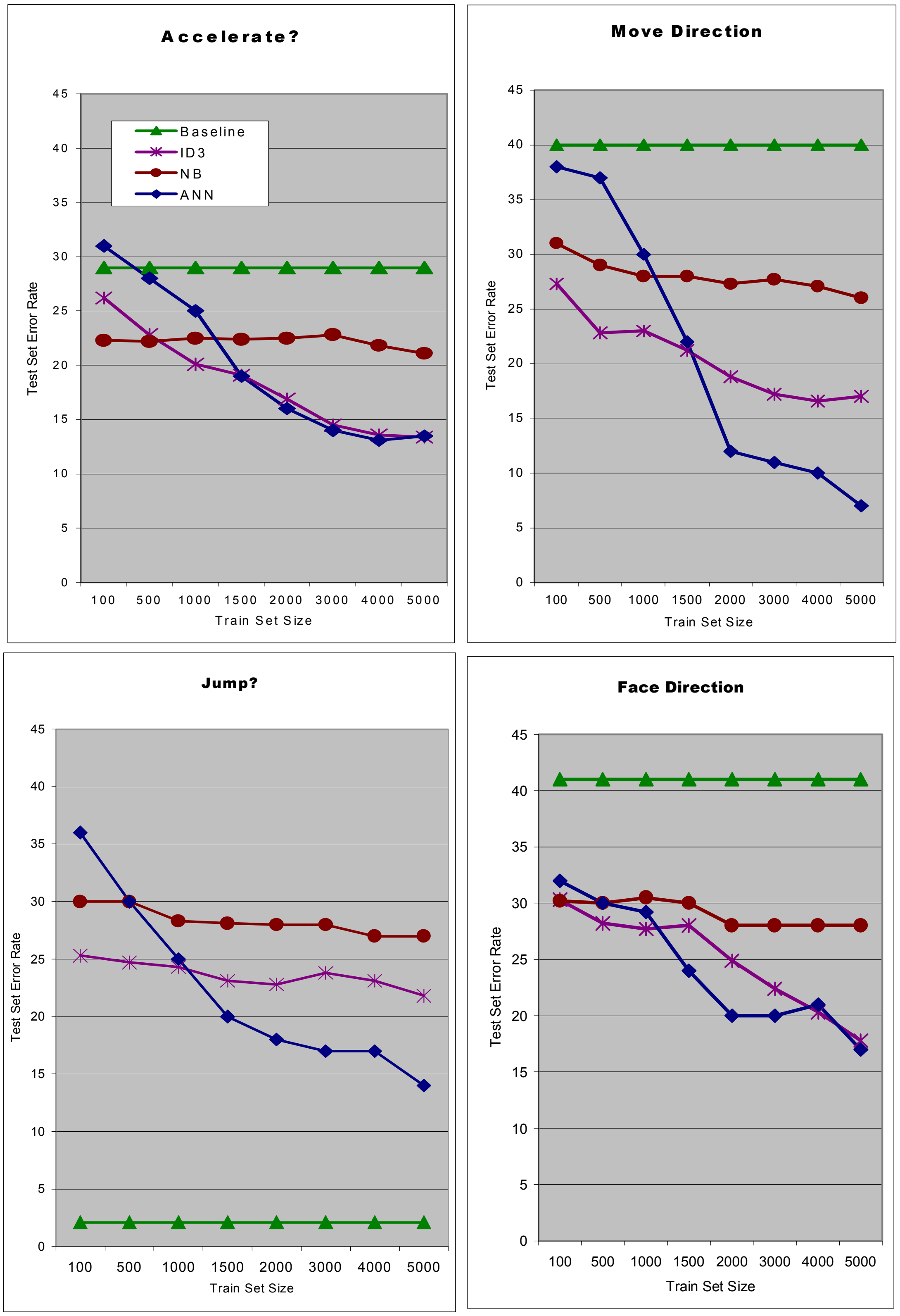

Figure 2: Error Rates of the Basic Learning Algorithms on the Four Tasks

Results Summary

It is helpful to simultaneously compare results of the various algorithms across all tasks. Figure 3 shows the error rate behavior of Naïve Bayes, decision tree and ANN on all four tasks.

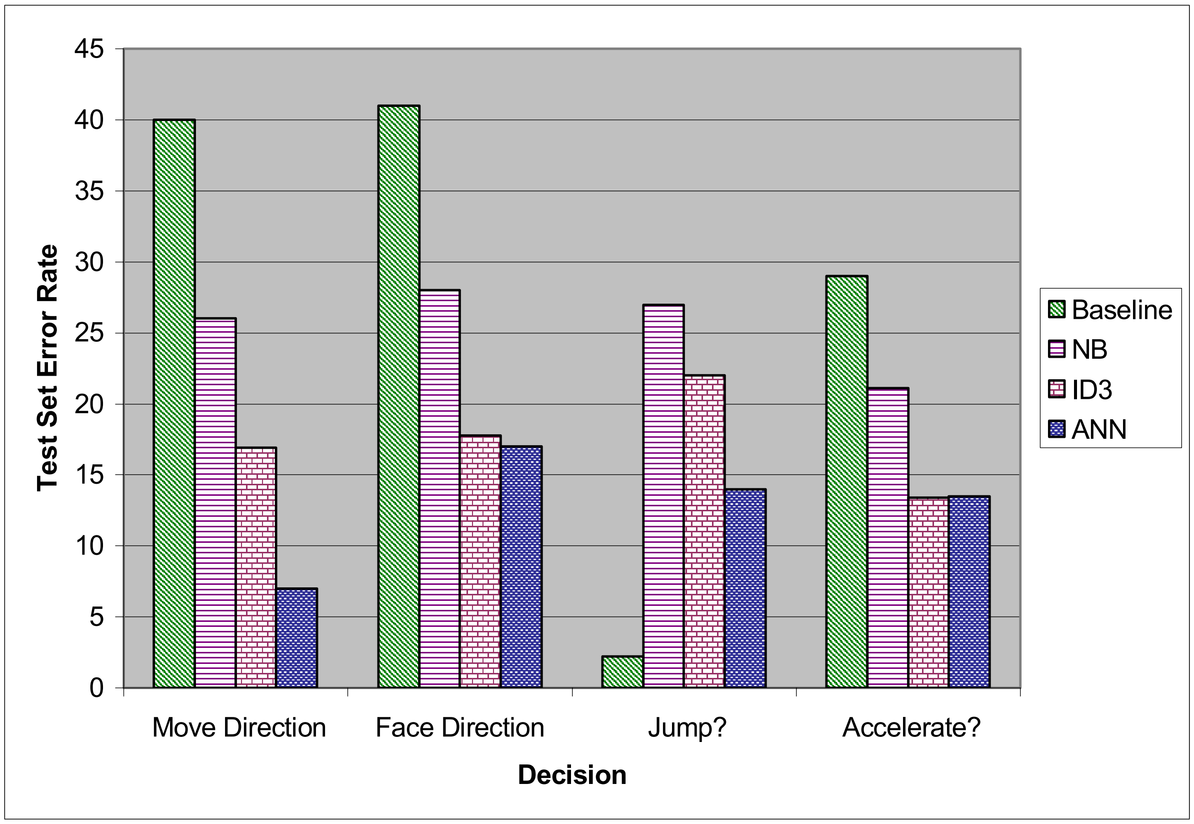

Figure 3: Test Set Error Rate Summary for ID3, ANN and Naïve Bayes

D. Discussion

Most of the four tasks saw diminishing returns on their error rate improvement after 3,000 training examples. However, for all three learning algorithms, the lowest error rates were achieved with a data set size of 5,000. With sufficient training data, the artificial neural network (ANN) always out performed the other algorithms. It's important to note that for smaller data set sizes the decision tree algorithm and Naïve Bayes actually performed better than the ANN. Since the FPS involves a continually changing domain, it is possible that any online learning would restrict the size of available training examples.

While, in general, the error rates were very low, one must be evaluate if they are acceptable for this domain. Each task must be considered individually, since the frequency of the acceptable scenarios under which these behaviors are performed varies greatly. Also it should be noted that the ANN requires many more CPU cycles than the Naïve Bayes or ID3 approach. In practice the ANN trained with 5000 examples took as long as 90 seconds. The speed of Naïve Bayes was essentially negligible. For this thesis we learned the player model offline and applied the results to the agents in the next game. However, if learning must be online and and speed is the primary concern, Naïve Bayes might be the best choice.

For *Face Direction*, the best performing algorithm (the ANN) achieved a 16% error rate. This means 16% percent of the time the agent will be facing a different direction than the expert player would have faced during that situation. At first this may seem low, but facing is crucial for fire fights; it is the difference between having a line of sight on the enemy and having his gun in your back. The classifier used allows an agent to have the incorrect facing one out of every six times; unfortunately this frequency is too noticeable.

Move Direction was classified very well by these learning algorithms, and received only a 6% error rate. *Move Direction* is probably the single most important task facing the agent. Goals are achieved by moving in the right direction, enemies are avoided by moving in the right direction and weaker enemies are overcome by moving in the right direction. Under the guidance of the learning algorithm the agent will decide correctly almost nineteen out of twenty times. This is clearly acceptable performance.

The learning algorithms achieved a 14% percent error rate on the *Jump* decision. However, the majority category always beats the learned prediction. One might wonder why not just guess the majority category since this would more closely model player behavior. However there is no drawback to jumping more often than is necessary. It is also a common tactic among some players. Furthermore, it's possible that other methods can help stabilize the performance on this task.

Accelerate is another important task. Knowing when to speed up can be the difference between securing the goal and allowing enemies to call in reinforcements. The best learner above has a 13% error rate on this function. Again, this is too high. It would be unacceptable if one in every eight times the agent fails to speed up to the goal or fails to speed up to avoid enemy fire. With each game having a short time limit and a great many of these situations, one in eight amounts to a behavior that will be noticed. This number needs to be at least cut in half; more advanced learning schemes must be considered.

IV. Ensembles: Boosting and Bagging

The basic learners performed well at classifying our data. With 5000 training examples the artificial neural network (ANN) in particular never got error rates worse than 16% on any of these tasks. However, as noted above this is still not acceptable. Since the agent will be using these decisions in real time again and again, the error rates need to drop. One thing that was noticed about the learned decision trees[7] and the behavior of the ANN is that a great many of the learned trees and functions work perfectly to account for many variables. However, it is the rare cases that hurt. What is preferred is a way to hedge the bet and rely on the functions that learn the task well, while somehow penalizing those that do not. Ensemble methods are one way to do this. An ensemble is a set of independently trained classifiers. The basic idea is to run the data on several different sets of data and have each learned function “vote” on the result. This vote then becomes the decision. Research has shown that usually ensemble methods produce more accurate results than singleton classifiers (Opitz & Maclin 1999). Bagging (Breiman 1996) and Boosting (Freund & Schapire 1996) are two popular methods for producing ensembles. These methods are fairly new and have not been tested on a domain similar to a first person shooter video game. Each of these ensemble methods was investigated in this thesis.

[7] See Appendix A for some examples.

A. Ensemble Algorithms

Bagging

A basic bagging scheme was implemented in this paper. The idea of bagging is to vary the training set several times and train one classifier for each data set. The pseudo code is described in table three.

Table 3 : Bagging Pseudo Code

```
Let $n = 5000$, the number of instances in the training data
Let $j$ be the test set instances (1000 instances).
For $t = 5$ to 30, counting by 5
    For $t' = 1$ to $t$
        Sample $n$ items from the original $n$ training examples with replacement, call this set $m$.
        Apply the learning algorithm to set $m$.
        Judge the decision on each member $i$ in the test set $j$, store result in $R(i, t')$.
    For each test item $i$ in $j$:
        Classify $i$ according to the most commonly predicted class recorded in $R(i, 1) \ldots R(i, t)$
```

With the exception of Naïve Bayes, bagging was rather beneficial. As shown in Figure 4, there is as much as a 3 percent increase in both the decision tree (ID3) and ANN. After the number of ensembles rose to 15 for the ANN, the increase in accuracy began to stabilize. The accuracy of the ID3 began to stabilize at ensembles of size 20. By way of comparison, the decision tree benefited slightly more from bagging than ANN. However, in all cases the ANN still outperformed the decision tree.

Boosting

Instead of a voting scheme (as in bagging), an algorithm known as boosting can be used to help weight misclassified and classified examples from within a training set. The weighted values can then be used to resample the training set and relearn the missed examples. The tuning set can be set aside as a means to gauge when to stop. We use a derivation of the original AdaBoost algorithm (Freund 1995). This version of AdaBoost was shown to be mathematically equivalent to Freund's original approach (Bauer, Kohavi 1999). Pseudo code follows in Table 4.

Table 4: Boosting Pseudo Code

```
Input:  Training set S of size m, where each example i is a a <x'_i,y'_i> pair
        Artificial Neural Network ANN
        Integer T (number of trials)

Set all instance weights to 1/m.
For i = 1 to T do
        $C_i$ = ANN(S)                          (The classification of test set S using the ANN)
        Compute weighted error on training set :
                $e_i = 1/m \sum_{x_j \in S : C_i(x_j) \neq y_i} weight(x)$       (Weighted error on training set)
                                                (Where $x_j$ is a training set instance that was
                                                misclassified)

        For each example $x_j$ in S' :
                If misclassified,
                        $weight(x_j) = weight(x_j) / 2e_i$
                Else
                        $weight(x_j) = weight(x_j) / 2(1-e_i)$
```

In some boosting implementations there is a step of normalizing the weights. No normalizing step was used here because the proportion of misclassified instances is e_i and these instances get reduced by 1/2 e_i. The total weight of the misclassified instances after updating is ½ the original training set total. Furthermore, the total weights of the correctly classified examples will be half the original training set total.

Operation of the Boosting Algorithm.

To demonstrate the helpfulness of this use of the tuning set to increase accuracy, let's step through the boosting algorithm and assume our learner is ANN. Starting with a training set size of 5,000, the training set is split off into a train and tune set. The tune set will contain 500 examples and be used for early stopping. Using ANN on the *Accelerate* data set shows that the base inaccuracy on the tune set is 21%. According to the update rule these 1,050 examples will be updated to a weight of 2.38 each (update factor of 1/(2*.21)). The correctly classified examples will have a weight of 0.63 (1/(2(1-.21). These weights were used for resampling for the next trial.

On the next trial the ANN made some mistakes that previously were classified correctly. Subsequently, these five mistakes end up getting weighted much higher (504). However after the second round of learning, tune set error is already reduced to 19%.

The results of the boosting algorithm were very good. For the *Accelerate* task an accuracy of 95.5% is now possible on the test set. This is a full 10 percentage point increase from bagging! However, initial experiments indicate that the other tasks in this project gain less from the algorithm. *Jump* increases less than 1 percentage point, while *Move* and *Face Direction* increase in accuracy 4 percentage points each.

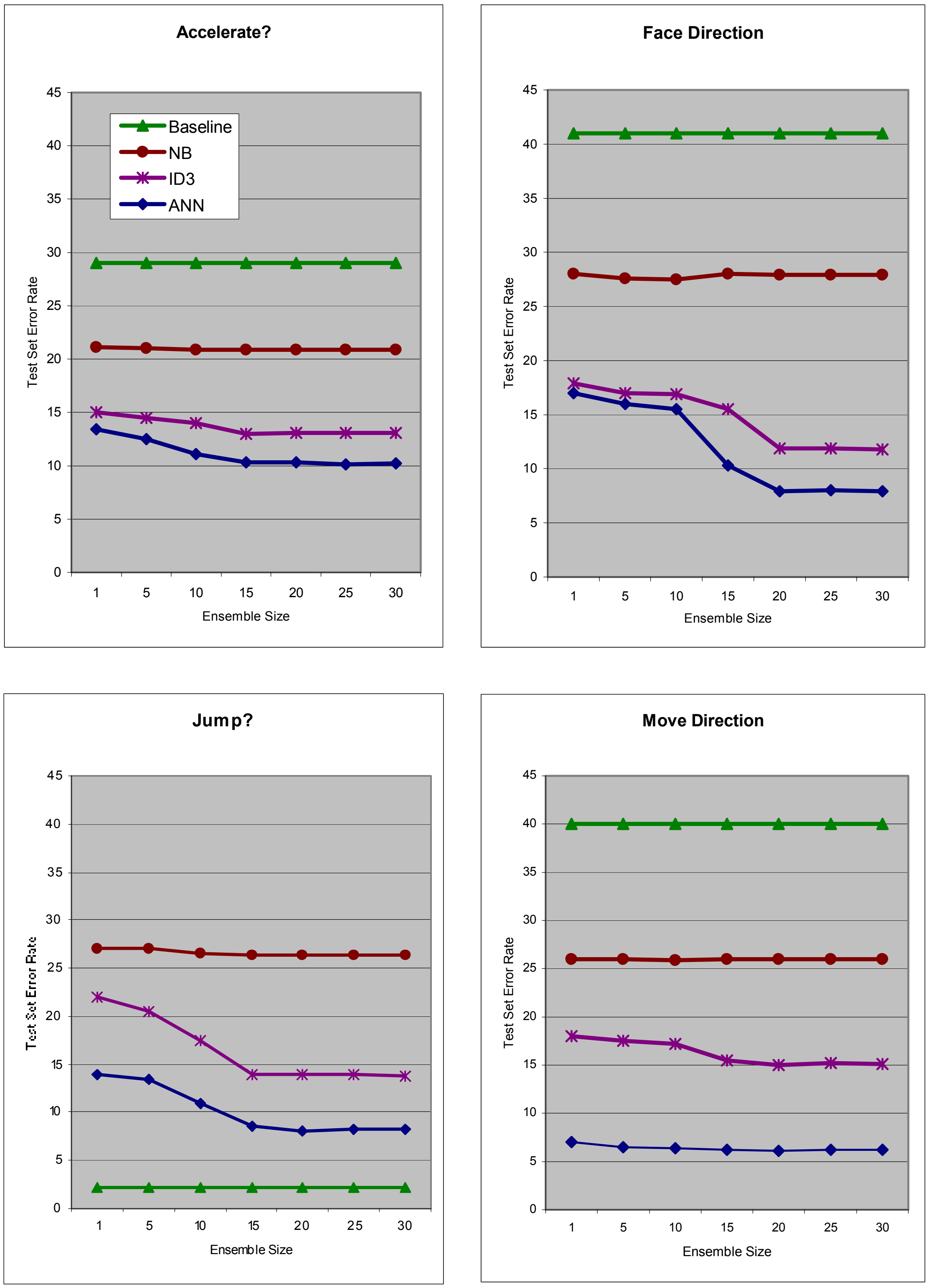

Figure 4: Error Rates of Bagging as a Function of Ensemble Size

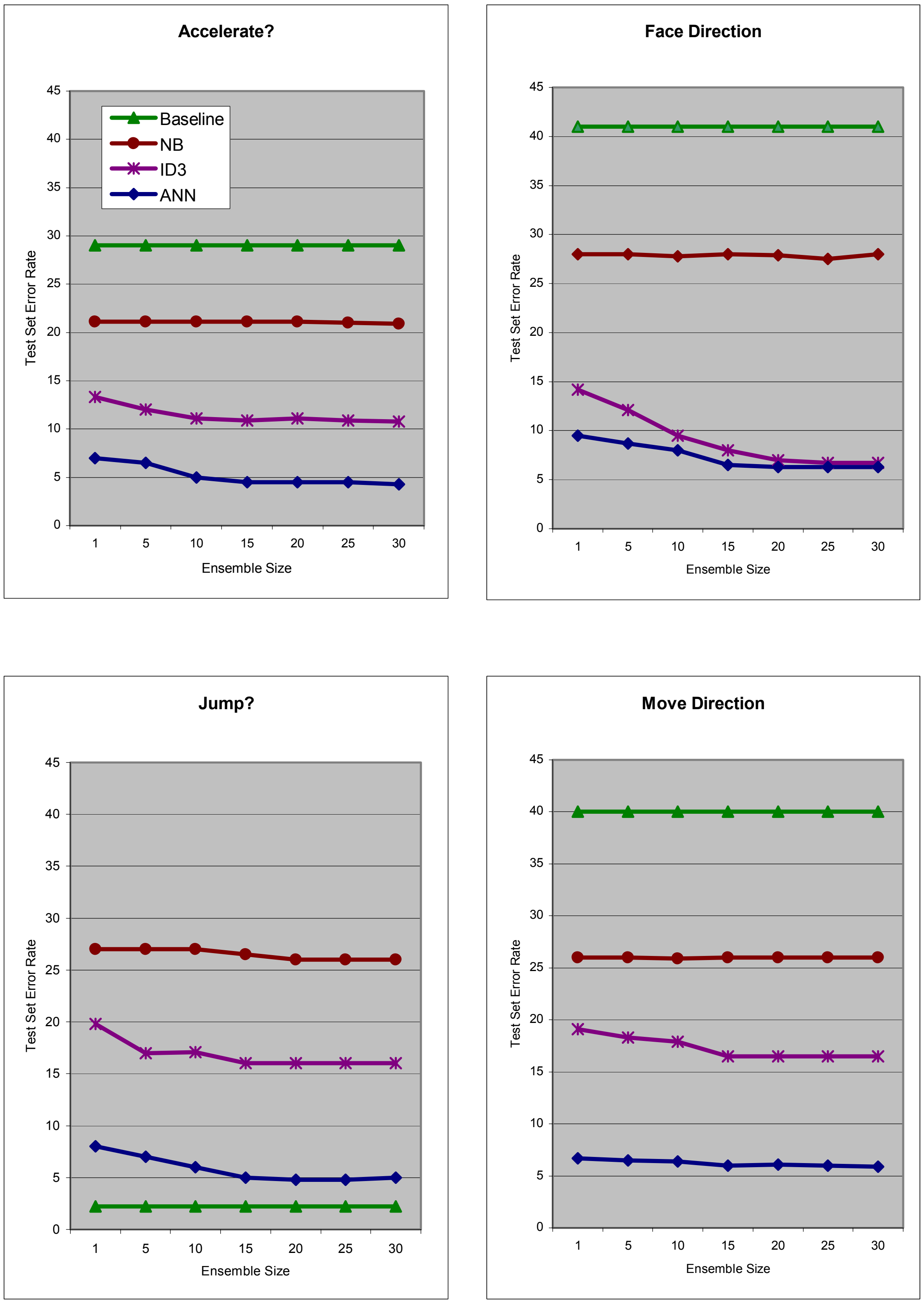

Figure 5: Error Rates of Boosting as a Function of Ensemble Size

B. Results of Ensemble Algorithms

The ensemble algorithms exhibited very good performance with even lower error rates. In general ANN paired with boosting always out performed ANN paired with bagging. The error rates were cut by as much half with boosting compared to training one ANN.

Finding the Right Ensemble Size

Some research on ensembles indicate that after the size of ensembles exceeds ten classifiers there is no further benefiting to increasing ensemble size (Hansen 1990). However, more recent work on the subject has shown that on some data sets increasing ensemble size to as high as 25 classifiers can still increase performance (Opitz 1999). Due to this project's unique domain and these differing findings, we will investigate several different ensemble sizes for our data set.

Figures 4 and 5 below show the results. After the number of ensembles rose to 15 for the neural network, the increase in accuracy began to stabilize. The accuracy of the decision tree began to stabilize at ensembles with 20 members. By way of comparison, ID3 benefited slightly more from bagging than the other classifiers. However, in all cases the ANN still outperformed ID3.

Test Set Error Rate as Function of Number of Training Examples

Tasks trained using only one ANN suffered from get diminishing returns after a couple thousand training examples, but for most of the tasks error rates continued to decrease for data set sizes nearing 5000. For *Face Direction*, the ANN kept achieving lower and lower error rates using boosting and bagging until data set size was increased to 5000. The boosting algorithm always outperformed the bagging algorithm. For *Move Direction* the ANN began to achieve lower error rates than the other two algorithms at 1500 data set examples. Accuracies of both ensemble methods began to level off around 2000 examples. The *Jump* and *Move Direction* decisions also continually decreased in error rates until the data set size increased to five thousand. Figure 6 shows respectable error rates for both boosting and bagging using the ANN as classifier. For the purposes of comparing the performance of learners with boosting with the performance of learners without boosting we will only show results of ANN, since ANN is the learning algorithm with overall best performance. Figure 7 shows an overall comparison of the performance of classifiers which use ensemble methods and those which did not.

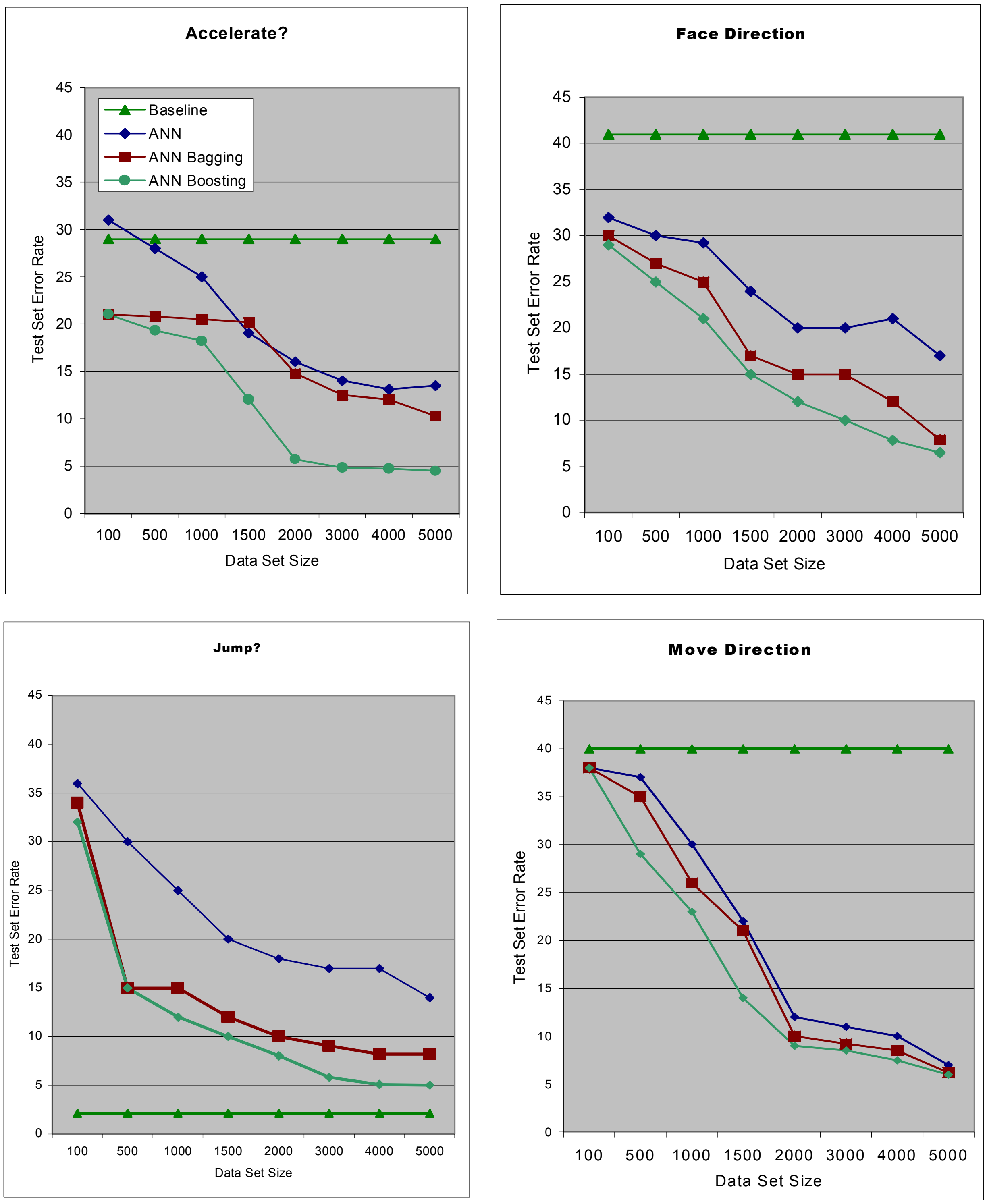

Figure 6 : Error Rates of ANN, ANN w/ Bagging, and ANN w/boosting

Confusion Matrices

A *confusion matrix* contains information about actual and predicted classifications done by a classifier. The data in a confusion matrix can be used to evaluate performance of classifiers. Confusion matrices for the four tasks were collected in addition to the error rates over various data set sizes. This data was collected over the test set and results are shown in Table 5.

Table 5: Test Set Confusion Matrices for the Four Tasks Using ANN with Boosting

Accelerate?

		Actual	
		Yes	*No*
Predicted	*Yes*	1213	170
	No	55	3562

Move Direction

		Actual	
		Yes	*No*
Predicted	*Forward*	1810	186
	Backward	54	2850

Face Direction

		Actual	
		Yes	*No*
Predicted	*Forward*	1523	475
	Backward	150	2582

Jump?

		Actual	
		Yes	*No*
Predicted	*Yes*	95	495
	No	5	4405

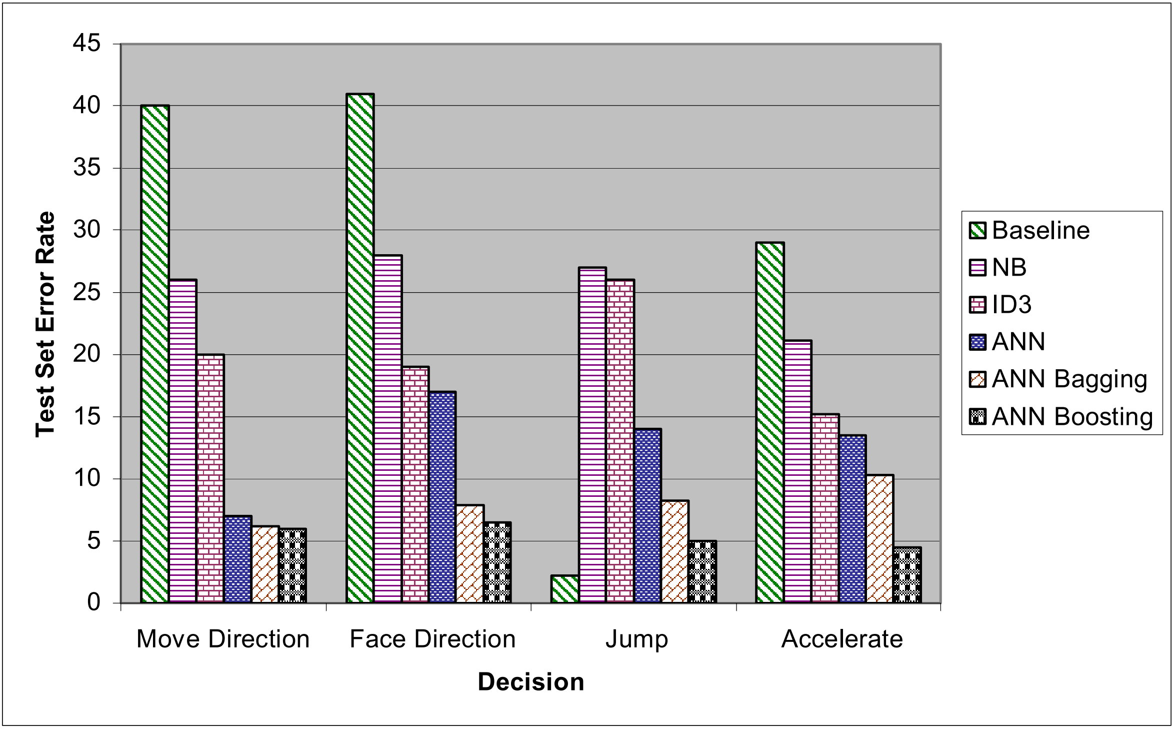

Figure 7: Error Rate Summary with Ensemble Methods Included

C. Discussion

Although all error rates dropped, certain tasks faired better than others when using ensemble methods. *Move Direction* is one of the tasks which showed little improvement using ensembles. While it did drop from a 7% error rate to 5.2%, this is a paltry improvement compared to the improvements in the other tasks. However, as seen earlier, *Move Direction* was already the best classified task and improvement was not necessary.

Face Direction fared well under the ensemble methods. With a single ANN the error rates were hovering at 16% but with boosting a major drop in misclassified results is shown. Boosting achieves a very respectable 5.3% error rate. As with Move Direction, this is an important task for the domain. The basic ANN algorithm would have made a mistake one out of every six times, which wasn't acceptable. One mistake in seventeen is much more acceptable, and while it will still be noticeable this error rate may be explained away as natural mistakes as opposed to obvious AI blunders.

As stated earlier, the *Jump* decision fares better overall when the majority category is guessed. This is still true with ensembles. In this case boosting got the error rate down to 5% while the majority category hovers around 2.5% percent. But is is important to consider what type of false negative statistics would be generated if it was always voted to not *Jump.* Table 6 shows that there are indeed more false negatives in the baseline case.

Table 6: Confusion Matrix for *Jump* Baseline

		Actual	
		Yes	*No*
Predicted	*Yes*	0	0
	No	126	4874

In this domain, it is desirable to lower the false negative rate at the cost of raising the true negative rate. In a FPS, it does not matter if the agent occasionally jumps for no reason. But at the same time the agent should jump when appropriate. Does our most accurate learning algorithm (boosting with an ANN) perform with more true positives than false negatives?

Table 5 shows the ANN with boosting predicts jumping when appropriate with 95% accuracy. Minimal accuracy is gained on true positives and the false negatives increase dramatically. With this algorithm in place, a jump occurred on 13% of the cycles when the agent shouldn't have jumped according to the training data. For this application it will be suitable to lower this number but allow most of the false negatives to slip through. Perhaps as a post-processing step a single flip of a coin could be used to decide whether or not to follow the "jumping advice." On real data, this would bring the true positive rate down, but it would also bring the false positive rate down to 6.5%.

Even with bagging the *Accelerate* task was still recording a 10% error rate, which is not acceptable. However, once boosting was added the *Accelerate* task increased in performance, to the point where it's now the most accurately predicted function at only a 4.8% error rate! This might mean that in the bagging ensemble method a great many of the examples randomly sampled were still voting the wrong way. While, with boosting the tuning set could be used to more closely weight the results. This is an important contribution, since it means the *Accelerate* task will err only one in twenty times. For such an important function, the lowest possible error rates are desired.

V. Related Work

In the domain of the FPS there is relatively little work on what learning algorithms work best. Some general frameworks for machine learning in video games have been laid out but not yet fully investigated. One framework was laid out by Zhimin Ding (1999). He proposes a design that is capable of real-time learning. The core algorithm to his system is the Markov decision process. Similar to our work, his first steps focus on transferring game data to applicable feature sets. He describes how such data can be collected using the example of a computer boxing game. Zhimin created a system to analyze the physical movements of a boxer (player) this system fills in the state transition probability matrix that is fundamental to the Markov decision process learning model.

Much of the work done by Zhimin in creating his learning system has exact equivalents to this thesis. In addition to describing a specific application of machine learning to a boxing game, he presents a general framework for embedding learning systems within games. Our approach departs from his framework when he assumes that the agent has all available knowledge about world physics. He uses this assumption to motivate the need for a smart feature-selection routine. Certain types of video games may have access to all world information but agents in an FPS typically do not have all this information available. For example, information about the architecture of the room is not complete. We have shown that with a good choice of features, learning can be effective without complete world information. For example, we showed that enemy locations can be split up into abstract “sectors” instead of exact room locations.

Another AI framework known as SOAR was developed by John Laird and his colleagues (Laird et al, 1986). This system creates an architecture for AI in the quake II (FPS) world. It uses reactive planners to simulate human behavior in a FPS (Laird 2000). Underlying the planner is a system that holds all the information about the conditions within the simulation. This approach depends on building an internal representation of the domain. The programmers specify a set of operations within the FPS (e.g. attack, wander, collect-powerup, explore). The agent predicts what the enemy will do based on environmental conditions. Based on these predictions the agent then uses a set of learned rules to determine a strategy. No learning is involved in this system, the behaviors are determined a priori by the creators of the agent. In this case, it’s the programmers who are the experts. Player modeling is not part of the process. In practice the system discussed in this paper would make similar choices. However, in our system the choices would be made based on functions created by one of the learners.

There has been a great deal of work done on types of ensemble classifiers. A recent summary can be seen in Chan, Stolfo Wolpert (1996). A variant of AdaBoost (Freund Schapire 1995) was used in this paper. Specifically we use the AdaBoost algorithm as adapted by Bauer and Kohavi (1999); the approach they take is simple and equivalent making it easy to insert into our FPS domain. Breiman describes a related method known as Arc-x4 (Breiman 1996) . Arc-x4 assigns fixed equal voting weights for each expert, and reweights exemplars using a much simpler function of the number of mistakes made by prior experts. Breiman has shown that under some situations Arc-x4 performs as well as the more complex AdaBoost.

An entire section of this paper was devoted to finding the right ensemble size. There have been many experiments on this issue. As mentioned above, Hanson has suggested that ensembles sizes greater than ten are not needed (Hansen 1990). It should be noted however that this work was done on very specific data sets. Other recent work does show that ensemble sizes up to 30 help reduce error rates (Opitz & Maclin 1999) (Cox 1999). In our experiments we found that sizes of 20 worked well.

VI. Future Work

Applying the ANN to the Game[8]

Applying an ID3 decision to a bot at run time is a tricky matter. Remember that a standard bot in a Quake3 game knows nothing about its surroundings. For example, it has no idea a priori that if it walks forward, it will hit a wall. This makes it difficult to directly apply the results to a real situation. This means some sort of back-end controller for the bot is needed to approximate what "back, forward, right, left, cover and shoot" means. Many games use some sort of node system to partition the playable areas for the bots (DeLoura 2000). This means that any time the bots are within a node they will know what nodes are in front of them, what nodes are for cover, what nodes are reachable from the given node, and what other enemies or players are in any given node. The figure below shows an example map. A circle represents a "node", a solid line is a connection and solid walls are shown by shaded polygons.

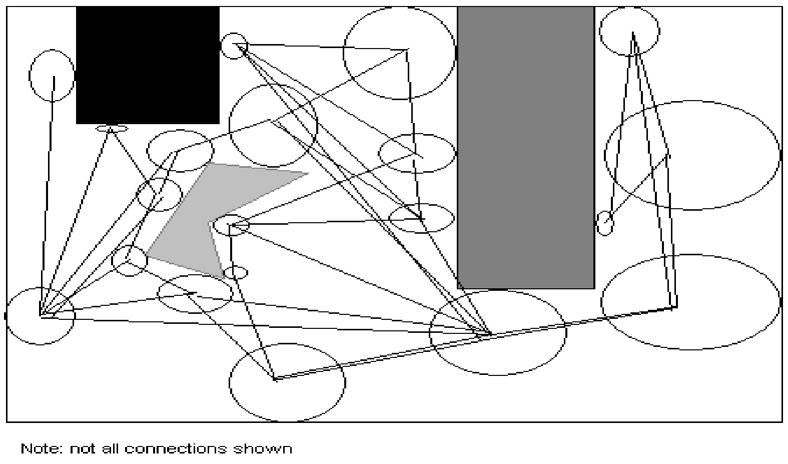

Figure 8: Sample Node System

Simple geometry can be used to derive what nodes are in front or back of each other in relation to the player. The height of walls is also known in an FPS. If a node intersects a wall at one height, but does not intersect higher up, this may be a possible duck point.

Finite States and Putting It All Together

A set of macro actions that can be performed at will is needed. For example, a routine to find a node that will give us cover from closest enemy, or a routine to move forward, towards the closest enemy could be used. Again, this is easy to do because there is a system in place for moving the agent about the world. Another reason some sort of controller is needed is that even if the AI tells the bot to attack, it might not know how to attack. What gun should the agent attack with, and what about grenades? All these things can be decided easily by some hand-coded rules. Each set of rule will be contained within a finite state Once the high level attack action is determined the finite states can then sort out the details.

[8] This thesis used version 0.13 of Soldier of Fortune 2.

Knowing how to attack is one example where a set of finer grained instructions is needed. But even a simple command to move forward isn't easy to interpret. Does the agent want to move the most directly forward, or off a little bit at an angle? This exact decision can also be made by some further rules. For example, perhaps it's better to move forward to spots that are close by. These kind of assumptions will need to be made while we're getting the learning system up and running. Perhaps someday these micro decisions can also be decided by player modeling.

In our system some of the authority is delegated to a smaller expert system, but in the meantime the classifier can control the more general behavior of fleeing, fighting, or holding our ground. This means that it will be more difficult for the player to spot a pattern. Patterns in FPS games often manifest themselves in how these higher level decisions are made. For example, if the player is there, the agent goes towards him. This unfortunately results in the player learning that course of action, and learning to account for it. However, now the basic direction of the bot can be modeled from player reactions. In practice this indeed holds to be true. Playing this simple combat map one can see agents deciding to run back for cover. Once the bots get to the desired spot the finite state takes over telling them the details of what to do next. For example, after running backwards because of the higher level decision the finite state may tell the bot to make sure and reload if their gun is empty. In general, this combination provides for a more dynamic game.

It has been shown that four ensembles of neural networks can be used to model player actions at any point in time. The accuracy on these is pretty decent, ranging all the way up to 96% accuracy for basic "move back" and "move forward" operations. It has also been shown that not all decisions in the game need to be learned. It's not necessary to have 100% accuracy here, but just to look intelligent and to present a challenge to the player. Many things remain to be done. For one the available actions should be extended to include crouching and various speeds of movement (e.g. run vs. walk). Also, finer grained decisions could be incorporated, i.e. the decision to shoot vs. throw a grenade. Since the actions taken do not have to be exact and it's in fact more enjoyable if it's dynamic, it might be interesting to try an algorithm with a bit more activity in the solution space. For example, genetic algorithms and their notion of occasional mutations may prove interesting.

VII. Conclusion

This thesis has explored several popular machine learning algorithms and shown that these algorithms can successfully be applied to the complex domain of the First Person Shooter (FPS). We performed our experiments on decision trees, naïve bayes classifiers, neural networks, and neural networks with boosting and bagging. We have shown that a subset of AI behaviors can easily be learned by player modeling using machine learning techniques. Under this system we have successfully been able to learn the combat behaviors of an expert player and apply them to an agent in the Soldier of Fortune 2™ FPS. The following tasks were learned: acceleration, direction of movement, direction of facing and jumping. We evaluate both empirically and aesthetically which learner performed the best and make recommendations for the future. We have created a system which uses these learned behaviors in a finite state system within the game at real time.

Since all the tasks shown in this project benefit to varying degrees from boosting, it is recommended to use boosting for the FPS domain. The artificial neural network (ANN) performed the best for any data set sizes over 2000. In practice it's most likely that learning will be done offline. So it's recommended to use an artificial neural network for computing these task decisions.

If learning is online, Naïve Bayes or ID3 with boosting may be a better option. This paper has shown that an ensemble size of 20 is a good choice.

As seen in the future work section there are many details to the first person shooter domain. A filter has been implemented in this project to transfer game data into training examples. Several learning algorithms have been implemented that learn crucial tasks well. But some work remains to efficiently allow game agents to make use of these classifiers.

VIII. References

Bauer, E. and Kohavi, R. (1999). An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine Learning*, 36(1-2): pp.105--139.

Breiman, L. (1996) Bagging predictors. *Machine Learning, 24*, pp.123-140.

Chan, P., Stolfo, S. and Wolpert, D. (1996), Integrating multiple learned models for improving and scaling machine learning algorithms. *Proc. AAAI Workshop*. AAAI Press, Portland, OR.

Cox, R. (1999), An investigation into the effect of ensemble size and voting threshold on the accuracy of neural network ensembles. *Proc. Australian Joint Conference on Artificial Intelligence,* Springer.

DeLoura, M (2000) *Game Programming Gems.* Charles River Media, MA.

Domingos, P. and Pazzani, M. (1996), Beyond independence: conditions for the optimality of the simple Bayesian classifier, *Machine Learning: Proceedings of the Thirteenth International Conference*, Morgan Kaufmann, pp. 105--112.

Freund, Y. and Schapire, R. (1995) A decision-theoretic generalization of on-line learning and an application to boosting*, Computational Learning Theory, Lecture Notes in Computer Science 904*, Berlin: Springer, pp.23-37.

Freund, Y. and Schapire, R. (1996) Experiments with a new boosting algorithm*. Proceedings of the Thirteenth International Conference on Machine Learning,* 148-156 Bari, Italy.

Hansen, L. (1990) Neural network ensembles. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12, pp.993-1001.

Laird, J.E., Rosenbloom, P.S., & Newell. (1986) Chunking in SOAR *: Machine Learning* 1(1), 11.

Laird, J.E. (2000) Adding Anticipation to a Quakebot. *Papers from the 2000 AAAI Symposium Series: Artificial Intelligence and Interactive Entertainment*, Technical Report SS-00-02. AAAI Press.

Liden, L. (2001) Using Nodes to Develop Stategies for Combat with Multiple Enemies In Artificial Intelligence and Interactive Entertainment*: Papers from the 2001 AAAI Symposium Series: Artificial Intelligence and Interactive Entertainment ,* Technical Report SS-00-02, 59-63. AAAI Press.

Mitchel, T. (1997) *Machine Learning*. McGraw Hill, New York.

Opitz, D. and Maclin, R. (1999) Popular ensemble methods: An empirical study*, Journal of Artificial Intelligence Research*, Volume 11, pp.169-198.

Perez, A. and Royer, D. (2000*) Advanced 3D Game Programming*. WordWare Publishing, Texas

Quinlan, J. (1986) Induction of decision trees*. Machine Learning*, 1(1), pp.81-106

Rosenbloom, P., Laird, J. and Newell, A. (1993) The Soar Papers *Readings on Integrated Intelligence*, MIT Press, Cambridge, MA.

Savitz, E. (2001) Video game industry poised for a shootout. CNN 1 May 2001 <http://www.cnn.com/2001/TECH/ptech/05/01/microsoft.video.game.idg/>.

Zhimin, D (1999) Designing AI engines with built-in machine learning capabilities. *Proceedings of the Game Developers Conference*. pp.191-203.

Appendix A

We now show portions of four decision trees (*Accelerate, Jump, Move Direction*, and *Face Direction*) learned by the ID3 algorithm. To conserve space the nodes of the trees are labeled with the following conventions:

Feature Name From Table 1	**Node Label**
Closest Enemy Health	EnemyHealth
Number Enemies in Sector 1	#EnemySector1
Number Enemies in Sector 2	#EnemySector2
Number Enemies in Sector 3	#EnemySector3
Number Enemies in Sector 4	#EnemySector4
Player Health	Health
Closest Goal Distance	ClosestGoal
Closest Goal Sector	GoalSector
Closest Enemy Sector	EnemySector
Distance to Closest Enemy	EnemyDistance
Current Move Direction	CurrentMove
Current Face Direction	CurrentFace

The edges represent the possible values for each feature. Some value ranges are not present to conserve space. The feature values *yes* and *forward* will be represented as 1, while feature values *no* and *backward* will be represented as 0. If a node ends with no further connections it is either a leaf node or a sub tree that is not illustrated. In the case of leaf nodes we list the number of occurrences at this node and the decision that was made. In the case of a sub tree we list the number of occurrences of each decision that was made in the abbreviated sub tree.

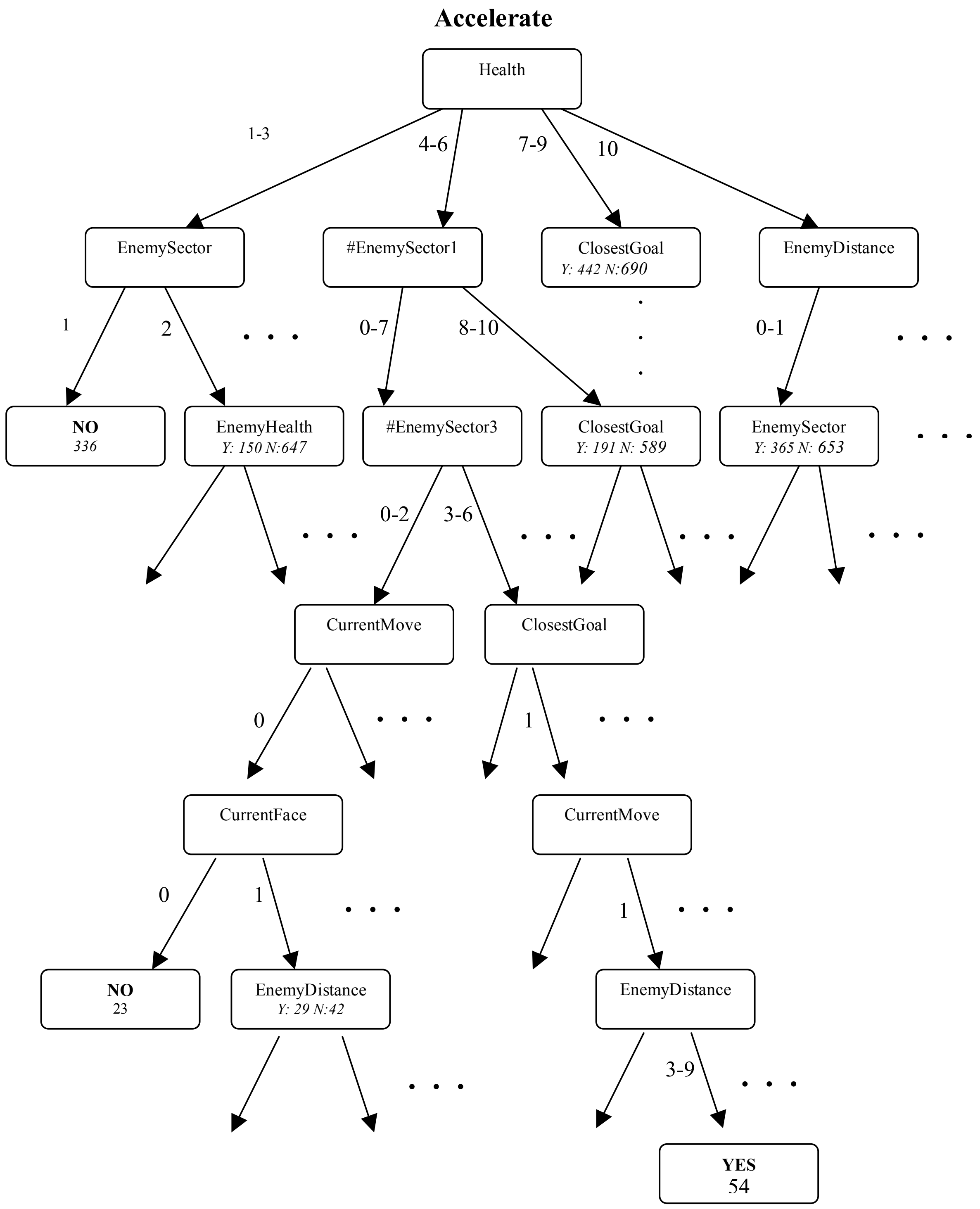

Figure 9: A partial decision tree as produced by ID3 on the *Accelerate* task

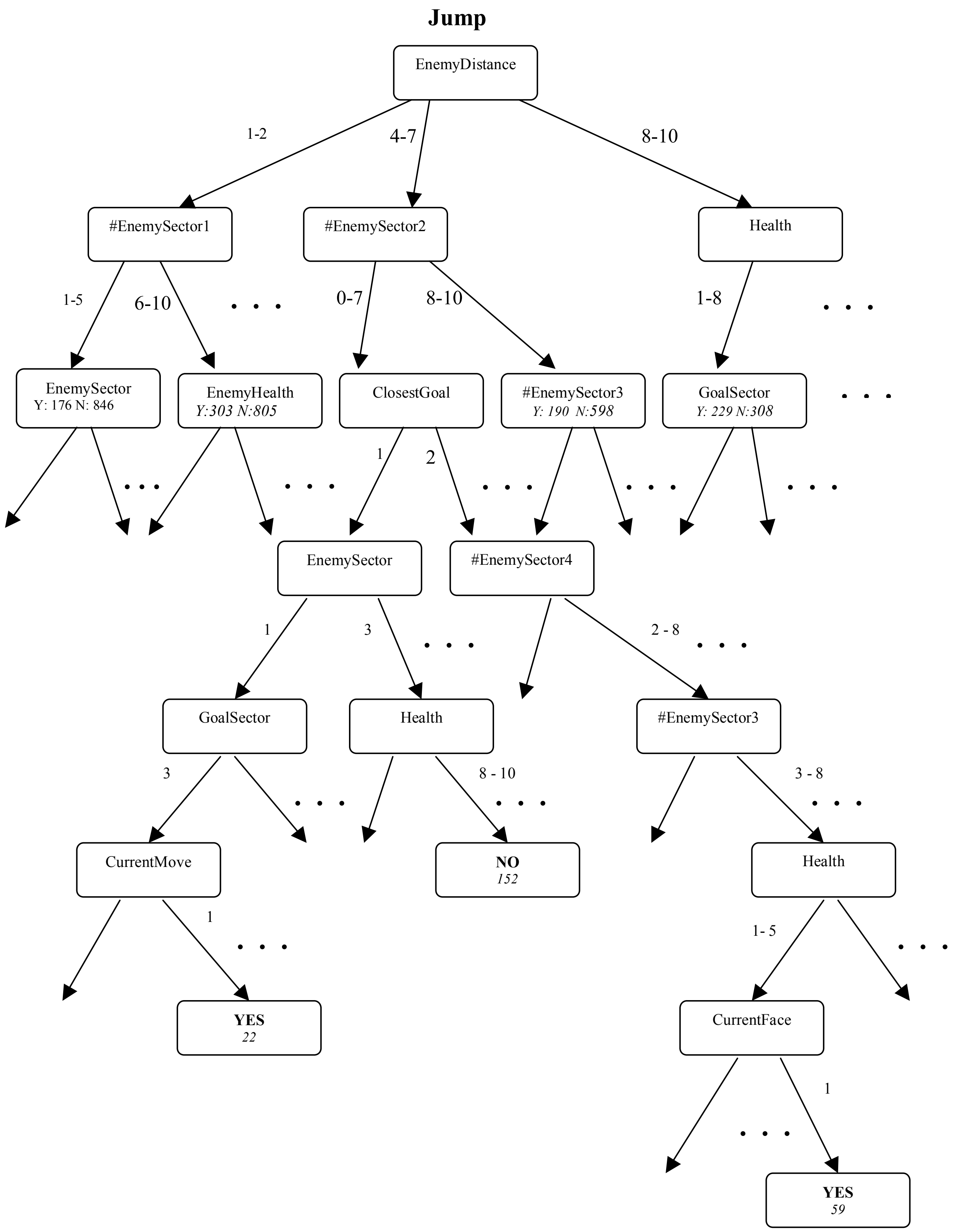

Figure 10: A partial decision tree as produced by ID3 on the *Jump* task

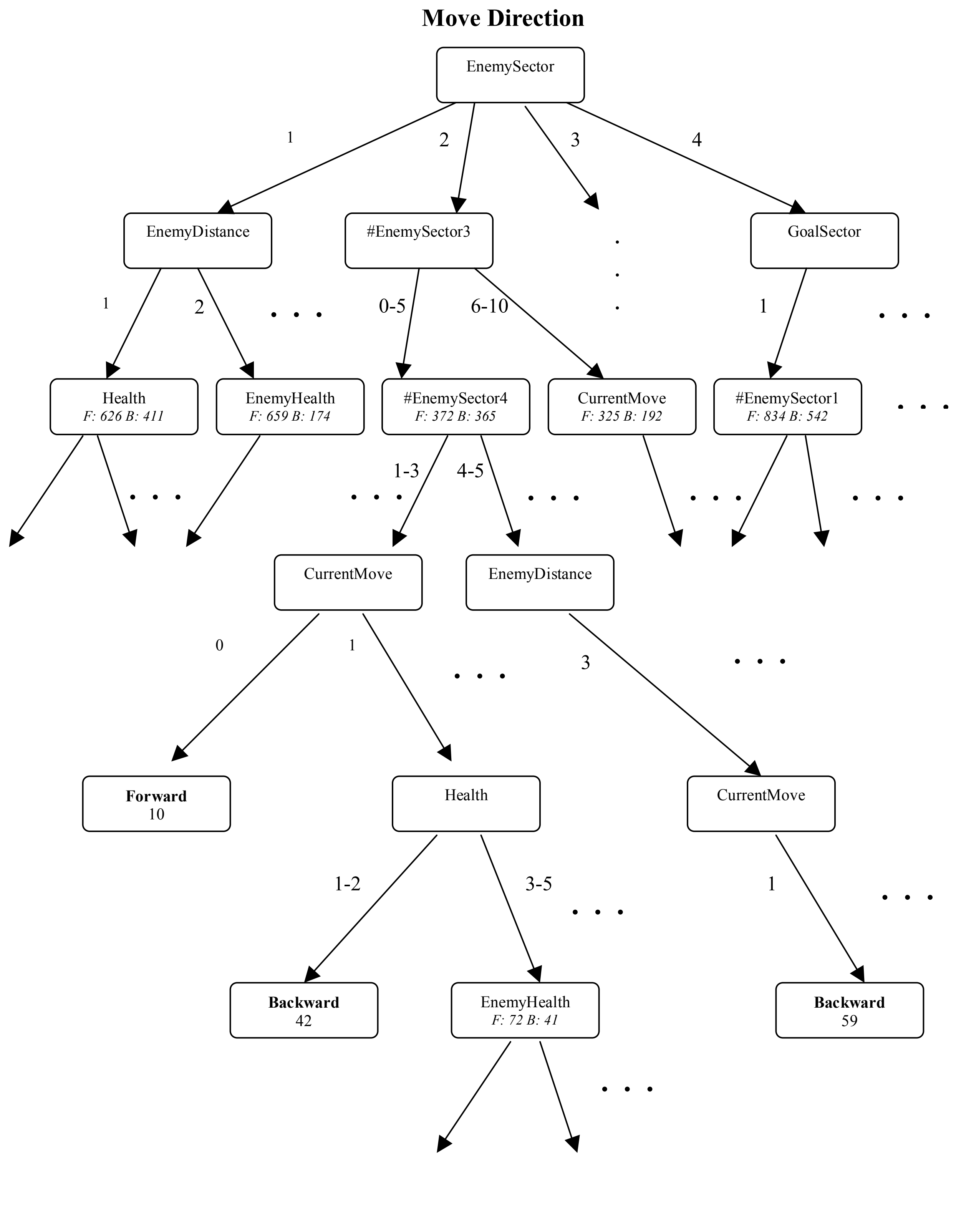

Figure 11: A partial decision tree as produced by ID3 on the *Move Direction* task

Face Direction

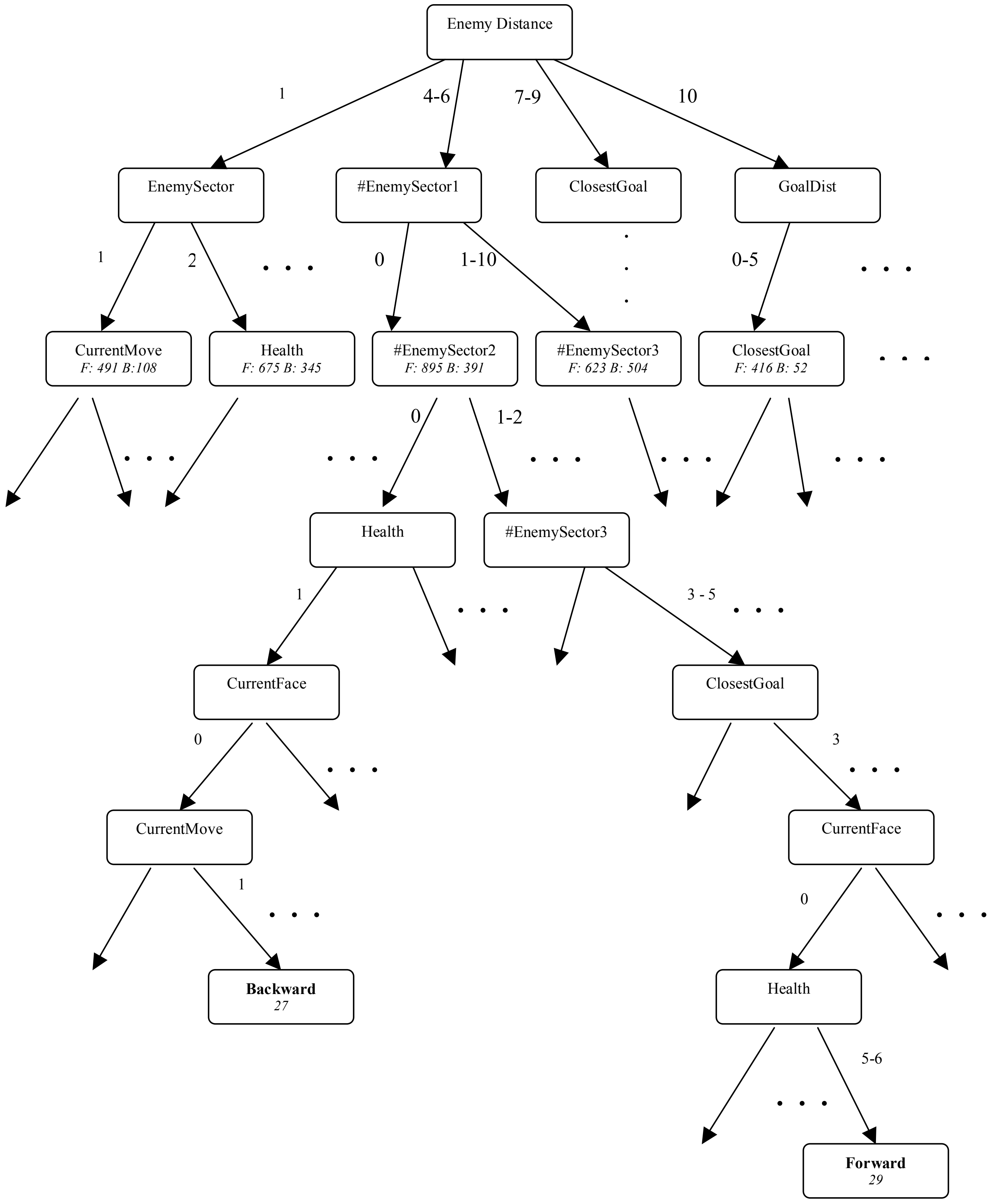

Figure 12: A partial decision tree as produced by ID3 on the *Face Direction* task