Integrated Machine Learning For Behavior Modeling in Video Games

Ben Geisler

Radical Entertainment
369 Terminal Ave. Vancouver, BC Canada
bgeisler@radical.ca

Abstract. In a multiplayer game, your opponents are other human players. These players make mistakes. Mistakes and miscalculations provide opportunity for other players. The challenge (and ultimately the fun) which comes from a multiplayer game is the give and take that comes from human interaction. 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. To mimic the multiplayer experience of human vs. human combat typically involves a high amount of tuning for game balance. Because of the difficulty of the problem, most single player games instead focus on story and other game types. A perfect artificial opponent for a first person shooter has never been modeled. 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 to model a player 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 paper explores several popular machine learning algorithms and shows how these algorithms can be applied to the game. We show that a subset of AI behaviors can be learned effectively by player modeling using the machine learning technique of neural network classifiers trained with boosting and bagging. 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*. However, the learning system has the potential of being extended to many other game types.

1. 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. This has its own merits as a game type in and of itself. However, there is room for other types of games with more interactive and lifelike opponents. Recently developers began noticing this deficit and games such has Half-Life©, UnReal©, Halo© and Call of Duty©, and 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. The exploitation of these flaws should be an aspect of the game, however the game must also keep challenging skilled players. In order to account for different skill levels, auto-adjusting difficulty systems could be incorporated into the behavior patterns and choices of opponents (Pfeifer 2004). The focus of this paper is on using a machine learning system to model player behavior. However, any such system should be designed with enough overrides and hooks to ensure game balance.

To ensure quality opponents for the skilled player of a first person shooter we will now turn to 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 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 paper.

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 paper 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 2003). 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" ™ and "Creatures" ™.

2. 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. Even expert players of an FPS 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.

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 interesting for soldiers to jump over barriers at a certain location with certainty (Geisler 2003). For this reason we want 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 paper 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, without proper calculations a bot may accidentally bump into a wall or walk off a cliff. A separate motor control system will be responsible for this.

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. 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. 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.

The Feature Set

This paper investigates only 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. Perhaps the most important information in any combat scenario is to know where the agent's enemies are. 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.

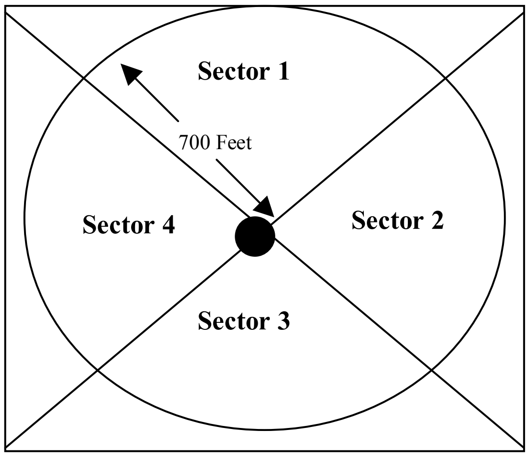

Figure 1: Game sectors with player centered in the middle.

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

[2] The AI agents in FPS video games are known as a "bots."

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.

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, most FPS games are 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[3].
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.
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) .

[3] 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 paper, 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

3. 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.

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).

```
Backprop(dataset, $e_{ta}$, $N_{in}$, $N_{out}$, $N_{hidden}$) Data in dataset is ($\mathbf{x}$, t), $e_{ta}$ is the learning rate, $N_{in}$ is the size of the input layer, $N_{tout}$ of the output layer, and $N_{hidden}$ of the hidden layer
    • Create a feed forward network with $N_{in}$ inputs, $N_{out}$ outputs, and $N_{hidden}$ hidden nodes. Initialize each $W_{ij}$ to some small random value
    • Until error value is below some threshold, repeat:
    • For each ($\mathbf{x}$, t) in dataset:
            • Input the instance $\mathbf{x}$ and compute outputs $o^u$ for every node (Forward propagation of activation)
            • Propagate errors backward through the network
                • For each output unit $k$, compute its error term: $delta^k = o^k(1 - o^k)(t\_ - o^k)$
                    • For each hidden unit $h$, compute its error term: $delta^h = o^h(1 - o^h)(\Sigma_{k \subseteq outputs})$ of $W_{kh}$ $delta^k$
                    • For each $W_{ij} = W_{ij} + delta\text{-} W_{ij}$, where $delta\text{-} W_{ij} = e_{ta}\, delta^j x^{ji}$
```

Table 2: Backpropogation Algorithm

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.

4. Experimental Methodology

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.[4]

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.

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

5. Ensemble Methods: Boosting

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. A great many of the 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 paper.

6. Results

Figure 3 shows that the error rates of using an artificial neural network with bagging were much lower than any other learning methodology we tried (Geisler 2002).

The *Move Direction* task was learned with only a 5.2% error rate. *Face Direction* had only a5.3% 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.

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 3: Confusion Matrix for *Jump* Baseline

		Actual *Yes*	**Actual** *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!

7. Integration of the Learned Model

Applying the ANN to the Game

Applying an ANN decision to a bot at run time was 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. We created a back end controller for the bot to interpret the semantics of "back, forward, right, left, cover and shoot. Many games use some sort of node system to partition the playable areas for the bots (DeLoura 2000) (Geisler 2003). 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.

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

Our finite state machine is capable of performing a set of micro-actions. For example, we created a routine to find a node that will give us cover from closest enemy and routines to move

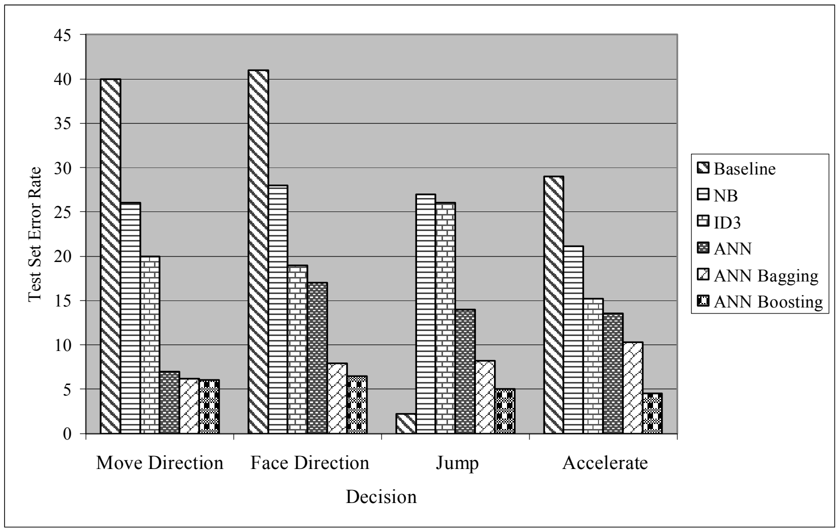

Figure 3 : Error Rate Summary

forward, towards the closest enemy could be used. This task was made easier by a modular design to the AI architecture (Ramsey 2003). What gun should the agent attack with, and what about grenades? All these things can be decided easily by some hand-coded rules and deferred to a separate module. 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.

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.

8. Conclusion

This paper has explored several popular machine learning algorithms and shown that these algorithms were successfully applied to the complex domain of the First Person Shooter (FPS). 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[5]. 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.

However, this is just the tip of the iceberg. First Person Shooters are not the only type of genre that can benefit from learning and modeling player behavior. Our work is directly applicable to any game genre that demands reasonably intelligent behavior from it's enemies. The machine learners presented in this paper were shown to be adept at modeling an expert player. However, when we used our machine learners to model an inexperienced player we ended up with an inexperienced bot. Therefore, this system could be used for fodder type enemies as well. Indeed, all the designer must do is "step into the shoes" of the enemy he is creating and act out a set of behaviors. With enough training and tuning this learned model could be directly applied.

In addition to enemy behavior modeling, this system could easily be extended to co-op games and games with non-human buddies. Machine learning in this context can be under the hood and transparent to the player. The player can keep his mind on having fun and not on training his buddies.

9. References

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

Freund, Y. and Schapire, R. (1996) Experiments with a new boosting algorithm. *Proceedings of*

[5] A heavily modified version of Soldier of Fortune 2 was employed for these experiments.

the Thirteenth International Conference on Machine Learning, 148-156 Bari, Italy.

Geisler, B. and Reed, C. (2003). Jumping, Climbing, and Tactical Reasoning: How to Get More Out of a Navigation System. *Game AI Programming Wisdom :* Charles River Media, MA.

Geisler, B. (2002). An Empirical Study of Machine Learning Algorithms Applied to Modeling Player Behavior in a 'First Person Shooter' Video Game. *M.S. thesis, Department of Computer Sciences*, University of Wisconsin Press. Madison

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.

Manslow, J. (2002). Imitating Random Variations in Behavior using a Neural Network. *AI Game Programming Wisdom*

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.

Pfeifer, B. (2004). Narrative Combat: Using AI to Enhance Tension in an Action Game. *Game Programming Gems 4:* Charles River Media, MA.

Ramsey, M. (2003). Designing a Multi-Tiered AI Framework. *Game AI Programming Wisdom 2:* Charles River Media, MA.

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