Adaptive AI for Fighting Games

Antonio Ricciardi and Patrick Thill
December 12, 2008
{aricciardi, kpthill}@stanford.edu

1 Introduction

Traditionally, AI research for games has focused on developing static strategies—fixed maps from the game state to a set of actions—which maximize the probability of victory. This works well for discovering facts about the game itself, and can be very successfully applied to combinatorial games like chess and checkers, where the personality and play style of the opponent takes a backseat to the mathematical problem of the game. In addition, this is the most widely used sort of AI among commercial video games today [1]. While these algorithms can often fight effectively, they tend to become repetitive and transparent to the players of the game, because their strategies are fixed. Even the most advanced AI algorithms for complex games often have a hole in their programming, a simple strategy which can be repeated over and over to remove the challenge of the AI opponent. Because their algorithms are static and inflexible, these AIs are incapable of adapting to these strategies and restoring the balance of the game.

In addition, many games simply do not take well to these sorts of algorithms. Fighting games, such as *Street Fighter* and *Virtua Fighter*, often have no objectively correct move in any given situation. Rather, any of the available moves could be good or bad, depending on the opponent's reaction. These games reward intuition, good guessing, and the ability to get into your opponent's mind, rather than mastery of abstract game concepts. Hence, the static AI algorithms that are most prevalent tend to do even worse in these games than in others.

These two problems are perfect for applying the techniques of machine learning. A better AI would have to be able to adapt to the player and model his or her actions, in order to predict and react to them. In this paper, we discuss how we developed such an AI and tested its effectiveness in a fighting game.

2 Goals

Our original goal was to produce an AI which learned online, allowing it to adapt its fighting style mid-game if a player decided to try a new technique. Obviously, this is preferable to an offline algorithm that can only adjust its behavior between games using accumulated data from previous play sessions. However, such an AI would need to be able to learn a new strategy with a very small number of training samples, and it would need to do so fast enough to avoid causing the game to lag. Thus, our first challenge was to find an algorithm with the learning speed and computational efficiency necessary to adapt online.

In addition, there are several other general requirements typical for online learning algorithms in video games [2]. For example, since the goal of a video game is to entertain, an AI opponent should exhibit a variety of different behaviors, so as to avoid being too repetitive. Furthermore, the AI should not be too difficult to beat; ideally it would adjust its difficulty according to the perceived skill level of the player. However, such modifications can be made relatively easily once the algorithm is capable of winning [3], so we chose not to focus on difficulty scaling for this project.

Finally, because players tend to have unique play styles in fighting games, it makes sense to develop separate models for each player. A game should be able to recognize a player and "swap in" the old model for that player, rather than learn from scratch each game. Ideally, when a player is found who cannot be confidently classified as one particular stereotype, the game would be able to merge the most similar models, allowing the opponent to begin the game with a strategy much better than that of a generic AI.

3 Early Implementations

Because implementing an algorithm for a complicated fighting game is very difficult, given the number of possible moves and game states at any point in time, we designed our first algorithms for the game rock-paper-scissors. This game abstracts all of the complexity of a fighting game down to just the question of predicting your opponent based on previous moves. As such, it was a perfect way to test a wide variety of algorithms quickly, and get a feel for which applied best to this problem.

3.1 Naive Bayes

For our implementation of naive Bayes, we used the multinomial event model, treating a series of *n* rounds as an *n*-dimensional feature vector, where each feature x_j contains the moves (*R*, *P*, or *S*) made by the human and computer in the jth round. For instance, if both the computer and human choose R on round j, then $x_j = <R, R>$. We classify a feature vector as *R*, *P*, or *S* according to the opponent's next move (in round $n + 1$). We then calculate the maximum likelihood estimates of $\varphi_{<R,R>|R}$, $\varphi_{<R,P>|R}$, $\varphi_{<S,R>|R}$, etc. , as well as φ_R , φ_P , and φ_S.

3.2 Softmax Regression

For softmax regression, we used the same feature vectors as in naive Bayes, with the same scheme for labeling. We derived the partial derivative of maximum likelihood function with respect to the ith -jth element of the *k*-by-*n* matrix θ and used it to update the matrix via gradient ascent. Due to the computational complexity of updating the θ matrix each iteration, we chose to run this algorithm offline.

3.3 Markov Decision Process

To apply reinforcement learning to rock-paper-scissors, we defined a state to be the outcomes of the previous *n* rounds (the same way we defined a feature vector in the other algorithms). The reward function evaluated at a state was equal to 1 if the nth round was a win, 0 if it was a tie, and -1 if it was a loss. The set of actions, obviously, consisted of *R*, *P* and *S*. We estimated the probability of transitioning from one state to another on a given action by keeping counts of all previous transitions. The optimal policy was calculated using value iteration.

3.4 N-Gram

To make a decision, our *n*-gram algorithm considers the last *n* moves made by each player and searches for that string of *n* moves in the history of moves. If it has seen those moves before, it makes its prediction based on the move that most frequently followed that sequence.

3.5 Algorithm Comparison

To test our algorithms, we wrote a Java program which simulated a game of rock-paper-scissors between two opponent AIs. Each learning AI was paired up with several static AIs that we designed to simulate real players. These static AIs would execute randomly chosen patterns whenever they saw a new state (where states were defined similarly to those of the MDP). For example, given the previous 5 rounds, one such AI might have a 30% chance of following with the pattern *R-R-P*, then executing a different pattern based on whichever new state it arrived in. Since our simulation allowed us to generate an arbitrary amount of data, we performed a simple cross validation on all the algorithms to see which performed best. We found that naive Bayes had a slightly better prediction rate than the other three algorithms but had a slower learning speed than reinforcement learning. We rejected softmax regression due to its slow computational speed and the fact that it did not outperform the other algorithms in any other regard. In the end, we went with the MDP, since we believed its learning speed gave it the most promise as an online algorithm.

4 Final Implementation

For the actual fighting game, we decided to implement our own game in Java rather than attempt to develop an algorithm for an existing open-source game. We did this for two main reasons. First, finding an appropriate open-source game and learning its code base would have taken significantly longer than designing a simple game of our own. Second, developing our own game gave us complete control over both the number of possible game states and the set of available actions, which made constructing the MDP much easier. Of course, the cost of this decision is that our game is much more rudimentary than a commercial fighting game. However, we believe our findings are still very applicable to larger-scale games.

In our simplified fighting game, players have access to a total of 7 moves: left, right, punch, kick, block, throw, and wait. Each move is characterized by up to 4 parameters: strength, range, lag, and hit-stun The moves interact in a similar fashion to those of rock-paper-scissors: blocks beat attacks, attacks beat throws, and throws beat blocks, but the additional parameters make these relationships more complicated. The objective is to use these moves to bring your opponent's life total to 0, while keeping yours above 0. Though most commercial fighting games tend to have a wider range of attacks available (typically around 20), as well as a second or third dimension for movement, their underlying game mechanics are generally very similar. In making the game, we tried to keep it simple enough to avoid spending too much time on aspects of the game unrelated to machine learning (such as balancing the attacks and finding appropriate animations for all the moves), while keeping it complex enough to make the problem challenging and make the algorithm applicable to commercial games.

The most difficult decision in porting our reinforcement learning algorithm to an actual fighting game was how to define a state. Our first idea was to include a history of the most recent moves performed by each player, as we did with rock-paper-scissors. However, since each player now has 7 available moves each game frame, this resulted in $7^{2 * \text{MEMORY}}$ possible states, where MEMORY is the number of previous frames taken into account. Having this many states caused a very slow learning speed, since an MDP needs to visit a state many times before it can accurately estimate the transition probability distributions for that state. In addition, we found that very similar states, whose histories differed in only one move by a single player, were treated as completely different states, so that the MDP would often behave inconsistently. Of course, when we tried decreasing MEMORY to reduce the total number of states, we found that the states did not contain enough information to find the correct action.

To minimize the number of states without losing too much information, we decided to ignore the specific moves made by each player and instead focus on the game statistics which resulted from those moves. In particular, we added information about the life totals, lag and position of each player. To eliminate redundant states, we extracted only the relevant information from these numbers. Specifically, we only kept track of the change in life totals from the previous frame, the distance between the two players, and the opponent's remaining lag. We ignored the AI's lag, since the AI can only perform actions during frames where its lag is 0. We defined the reward function for a state as the change in life totals since the last frame. For example, if the AI's life falls from 14 to 13, while the opponent's life falls from 15 to 12, the reward is (13 - 14) - (12 - 15) = 2.

To test the idea of recognizing a player based on his or her play style, we tried a few simple algorithms which rated the similarity of two MDPs. We compared the value function, transition probabilities, and number of visits for each state in the first MDP to those of the corresponding state in the second MDP. Our hope was that a new MDP playing against a particular opponent would resemble an MDP for the same player from a previous play session, and that this resemblance would be noticeable relatively early in the new play session.

4.1 Improvements to the Standard MDP

Since a cunning human player may vary his strategy mid-game, it is important give the most recent state transitions the most weight, since they are most indicative of the opponent's current play style. For example, suppose that the MDP has been playing against a player for several rounds, and in state S_1 it has taken action A 1000 times, 100 of which resulted in a transition to state S_2, yielding an estimated transition probability of $P_{A, S2} = 0.1$. Now suppose the opponent shifts to a new strategy such that the probability of transitioning to S_2 on A is 0.7. Clearly it will take many more visits to S_1 before the estimated probability converges to 0.7. To address this, our MDP continually shrinks its store of previous transitions to a constant size, so that data contributed by very old transitions grows less and less significant with each rescaling. In our example, this might mean shrinking our count of action A down to 50 and our counts of transitions to S_2 on A down to 5 (maintaining the old probability of 0.1). This way, it will take far fewer new transitions to increase the probability to 0.7.

In addition to increasing the weight of the most recent transitions, we needed to increase the rate at which the transition probability estimates were recalculated from the stored counts, so that the change in the AI's behavior would be noticeable quickly. This was particularly important for preventing the opponent from exploiting a newly discovered weakness in the AI's strategy for more than a few frames. After an initial exploratory phase, our algorithm performs value iteration on every frame. To encourage exploration of new moves after this point, it chooses a move at random from those within a tolerance of the value of the best move. Also, to prevent the algorithm from getting stuck in a local maximum, as well as to prevent repetitive behavior, we temporarily remove a move from the list of actions if it has been used three times in a row.

Since our goal was to develop a player model, rather than a general strategy for playing the game, we hard-coded a few rules that would always be correct, regardless of the opponent's strategy. For example, if your opponent is more than two spaces away, it is useless to block, since no attack has a reach greater than 2. These rules sped up the learning process by preventing the AI from wasting time exploring useless moves. To decrease the value of these actions, we created a dummy state with a very low reward, and gave each useless action a 100% chance of transitioning to this state.

5 Results

We tested the MDP AI at first against other static AIs, and then against human players. The static AIs were modeled by state machines that would execute random patterns associated with each state (as described in section 3). After making the aforementioned improvements to our algorithm, the MDP AI beat the static AIs nearly every game. Of course, this is not surprising, since the static AIs ran on the same states as the MDP and thus were easy for it to predict.

Against human players, the MDP was very strong, winning most games and never getting shut out. However, the only experienced players it faced were the researchers themselves. Also, while any AI has the advantages of perfect control, timing, and perception against human players, those advantages may have been magnified in this case by the game’s unpolished animations and slow frame rate. Nevertheless, this is a good result, and the success of the AI can be confirmed by other observations.

One goal of the project was to create an AI which could find its own way out of the simple patterns that players might use to exploit the AI. For example, once a player had backed the AI into the corner, he could simply use the kick attack over and over, with its long range preventing a jab or throw from being effective. The only way for the AI to recover from this situation is to step forward before jabbing or throwing. Over the course of user testing, the AI had to recover from a number of such exploits. On the average, it made a correct choice on the around sixth attempt, but it was subsequently very quick to counter if it saw the same tactic later on. In addition, we occasionally saw it develop more complex counters. For example, if the opponent was simply repeating an attack over and over, the MDP would sometimes learn to block and then counter while the opponent was in lag. Since these behaviors were not preprogrammed and had to arise from random exploration, we viewed this as a significant achievement. If the AI were able to learn for many games against the same player, rather than just the few rounds it got against our participants, these behaviors may develop more fully and even interact with each other.

Opponent recognition worked very well for the static AIs against which we first trained our MDP. Indeed, after 25 rounds the similarity between two instances of the AI trained against the same static AI was infinity, that is, the two instances had identical transition matrices, up to Java's rounding error. However, recognition grew spottier when applied to human players after 3 rounds of play. Some players consistently produced similar MDPs, whereas the MDPs of other players varied widely. It is unclear whether this is due to a flaw in the comparison algorithm, the relatively short play time, or the players themselves modifying their strategy between games, although it should be noted that their inexperience with the game makes the last option the most likely.

6 Conclusion

The AI algorithm presented here demonstrates the viability of online learning in fighting games. We programmed an AI using a simple MDP model, and taught it only a few negative rules about the game, eliminating the largest empty area of search space but giving no further directions. Under these conditions the AI was able to learn the game rules in a reasonable amount of time and adapt to its opponent.

This project also demonstrates several ways in which the domain of fighting games poses a unique problem. The extremely quick learning speeds and computational efficiency required for game AIs make several standard machine learning techniques, such as naive Bayes, impractical, and forced us to make modifications to the standard MDP algorithm. Indeed, pattern-matching in general is a poor metaphor for this domain, as fighting games often require one to recognize and adapt to a new tactic as soon as it appears, rather than when it came be recognized distinctly as a statistical pattern. One might expect that an AI with this property needs to be written specially for fighting games, but we have demonstrated that it can be achieved by adjusting a standard reinforcement learning algorithm to have a short memory and a rapid update rate.

Regarding player recognition, it is unclear based on the results whether it is feasible to recognize a player based on previous play sessions and swap in an old AI faster than it takes the new AI to learn from scratch. Perhaps player recognition would be more successful using a separate learning algorithm, rather than comparing the new and old MDPs with a formula. Of course, it may also be acceptable to have a player directly identify him or herself with a profile, so that the recognition step is unnecessary.

7 Future Work

An immediate improvement to the algorithm might be made by experimenting more with possible state compositions. For example, perhaps the MDP could learn to play more conservatively if it knew when its life total was near 0. In addition, it might be fruitful to separate the game playing and player modeling aspects of the project into two separate algorithms, so that general game knowledge gained in one encounter might more easily be transferred to another against a different player. A more significant next step would be to generalize the algorithm to be used in more complex games and see what problems result from the larger scale. Finally, since the algorithm presented here is only somewhat capable of recognizing players, it would be interesting to further explore the feasibility of reusing old MDPs. In addition to swapping in an old MDP for a returning player, this might allow a new player's strategy to be classified as being a combination of the strategies of previous players, whose respective MDPs could then be merged for the new player.

Acknowledgements

We would like to thank Professor Pieter Spronck of Tilburg University for offering advice during the early stages of this project.

References

[1] Charles, D et al. "Player-Centered Game Design: Player Modeling and Adaptive Digital Games". http://info200.infc.ulst.ac.uk/~darryl/Papers/Digra05/digra05.pdf

[2] Spronck, Pieter. *Adaptive Game AI*. http://ticc.uvt.nl/~pspronck/pubs/ThesisSpronck.pdf

[3] Spronck, Pieter. Personal interview. October 24, 2008.