Machine Learning for Auto-Dynamic Difficulty in a 2-D Space Shooter

Nick Cooper
nacooper@stanford.edu
http://www.stanford.edu/~nacooper

Abstract

This project explores the idea of using online reinforcement learning to allow a 2-D space shooter to adapt its levels based on a user's playing style. The "level-building agent" uses knowledge about the recent actions of the player as input, and attempts to learn parameters associated with generating enemies in the level, in hopes of maximizing a player's "enjoyment" of the game. As the player's skill increases, the maximum value of this "function" changes, and the agent must continue modifying its parameters in order to maximize this changing function.

Background

A major challenge in video game design is accommodating players of many different skill levels. Some games today do not even attempt this – they are either directed at casual gamers, or highly skilled "hardcore gamers." The problem with this is that either type of game will probably not be enjoyable to the other crowd. Hardcore gamers will be bored by a game that is too easy, and casual gamers will not be able to pick up a new game that is overly difficult. Many games try to accommodate a wider range of players by including a difficulty setting, which the player can manually adjust. However, there are also some problems with this approach. While playing through a game, players will naturally learn from their experiences, and become more skilled at the game, while possibly developing their own unique strategies and techniques. And, while this is happening, the difficulty of the game is remaining constant, and a single general strategy of the player can be used repeatedly with success. Even more importantly, players themselves are often bad at judging their own skill level, and can easily become bored or frustrated with a game based on a bad initial difficulty selection.

Some developers have tried to solve these problems with the idea of Auto-Dynamic Difficulty. For example, in the game Max Payne, the player's performance in a given level determines how strong the enemies will be in the next level. Variables such as enemy health and weapon strength are adjusted, in order to make the game easier or more difficult for the player, depending on his/her skill level. The health of the player and the number of times the player died in the level are used to determine how to adjust the difficulty.

Game Description

The game being used for this project is a 2-D space shooter called *Office Wing*, in which the player controls a paper plane and battles various office supplies. The player's viewpoint is from the side, and all control of the plane is in 2-D. The paper plane is continuously moving through the level, with the viewpoint scrolling to the right. Enemies appear in waves, and come from the right side of the screen. The goal of the player is to progress as far as possible before losing all of his/her lives. This can be done with or without defeating enemies, although defeating enemies makes this task easier by eliminating the current threats to the player. Enemies can be defeated by using several types of projectile weapons that are available to the player. The player dies when hit by an enemy or some other hazardous object, such as an enemy projectile. If the player has extra lives remaining, the paper plane immediately returns to the screen and the game continues. Otherwise, the game is over and the player has lost.

Differences from other Reinforcement Learning Problems

Several aspects of this problem make it somewhat different than other reinforcement learning problems. First, the function we are trying to maximize is "player enjoyment." This is not only a difficult function to define, but it is also changing over time as the player gains experience and skill. Also, since we are trying to learn online at a fast enough pace to keep a player interested, it is more desirable to make fast progress towards an optima than to make slow progress and eventually converge. In addition, if we are able to converge to the maximum of this function temporarily, it may not be a good idea to stay at the optimum. Even if the player is being challenged, it will not be fun to face the same exact challenge repeatedly.

Rewards, Inputs, and Parameters

Player enjoyment is modeled as a relation between the number of "close calls" and player deaths. Close calls occur when an enemy or other hazardous object passes within a defined area around the player. A high number of deaths indicate that the game is too difficult for the player, and as a result the player is probably not having fun. A low number of deaths and close calls indicate that the player is not being challenged, and may be losing interest in the game. What we seek is a high number of close calls and a low number of deaths, which would indicate that the player is being given a good challenge without being overwhelmed.

The level-building agent receives positive awards for close calls, and negative awards for player deaths. The agent also automatically receives a negative reward for each wave it sends at the player, which acts as a penalty for when there are very few close calls and no deaths in a wave.

The features of the state that are observable by the level-building agent are very few, consisting only of values indicating the player's total movement distance and most-used weapon over a recent time period. Larger sets of features were considered, but testing revealed that these two features were of far greater importance than the rest. Both parameters are based on recent or current time because a player's skill level and playing style are continuously changing over time. As a result, the agent should not take into account the player's actions that occurred a long time ago, as the player may have developed new skills and strategies since then.

The level-building agent attempts to learn and continue to update parameters in order to keep the "player enjoyment function" at a maximum. These parameters are related to the types, number, formations, and movement patterns of enemies to be sent out in a given enemy wave.

Reinforcement Learning Methods

The problem of learning the optimal action to take at a given state was posed as an MDP, with 12 possible states corresponding to different weapon preferences and movement amounts. The rewards and state transition probabilities were learned as gameplay progressed, and a slightly modified version of Value Iteration was used to learn the optimal value function and policy.

The basis for this was the Bellman Equations:

$$V^{\pi}(s) = \max_a \, R(s, a) + \gamma \sum_{s'} P_{sa}(s') \, V^{\pi}(s')$$

$$\pi(s) \quad = \operatorname{argmax}_a \, R(s, a) + \gamma \sum_{s'} P_{sa}(s') V(s')$$

After an action completed, the level building agent received information about the resulting state s' and the reward R(s, a) for the previous state and completed action. This information was used to update the reward estimates, the state transition probabilities, and then the value and policy at each state. Learning a model with state transition probabilities proved to be important, because it allowed the agent to predict the player's state resulting from an action. For example, if the player is primarily using the InkBlob weapon, it may be very likely that they will switch weapons if they encounter Whiteout enemies, who are resistant to the slowdown effects of the InkBlob.

One of the most important modifications that allowed learning to occur online at a reasonable speed was splitting up an action into four different components. Without this split, there would have been approximately 11250 possible actions at each state. An action was represented as a formation, a path type, a number of enemies, and one to three enemy types, with each component having 2-6 possible values. Each of these components was treated as an individual "action" and optimized independently. For example, if the level-building agent executed the action *{formation = linear, path type = vertical wave, number of enemies = 4, enemy type 1 = enemy type 2 = enemy type 3 = Pencil}*, then a portion of the reward received would be associated with each of the *linear, vertical wave, 4 enemies,* and *Pencil* component actions. A policy is associated with each component, and the overall policy takes the action components given by each component policy.

State transition probabilities were updated after each wave, with each $P_{sa}(s')$ being computed as:

$$P_{sa}(s') = \frac{\text{\#times took we action a in state s and got to s'}}{\text{\#times we took action a in state s}}$$

The reward updates were more complicated. Let a be the action taken at state s.

$$R(s, a) := R_{old}(s, a) * \alpha_1 + R_{observed}(s, a) * (1 - \alpha_1) \qquad 0 \le \alpha_1 < 1$$

where $R_{old}(s, a)$ is the previous estimate of the reward for taking action a at state s, and $R_{observed}(s, a)$ is the reward just observed for taking action a at state s. This is essentially a weighted average of the previous reward estimate and the newly observed reward. This takes into account the previously observed rewards and the most recently observed reward for a state/action pair, generally favoring the most recent reward. The value used for α_1 in this equation was determined after a great deal of experimentation.

This estimated reward update alone did not yield very desirable results. For example, if the player had been using the Spitball weapon for the entire game and suddenly switched to the InkBlob, suddenly the game's difficulty would drop significantly, because its reward estimates for these unseen states were still at their initial values. This problem was handled by also updating the reward estimates for unobserved states:

for each $s_i \neq s$

$$R(s_i, a) := R_{old}(s_i, a) * \alpha_2 + R_{observed}(s, a) * (1 - \alpha_2) \qquad \alpha_1 \leq \alpha_2 < 1$$

Each estimated rewards for action a under states other than s were updated very slightly. As a result, situations such as the weapon switch described above did not radically change the game's difficulty.

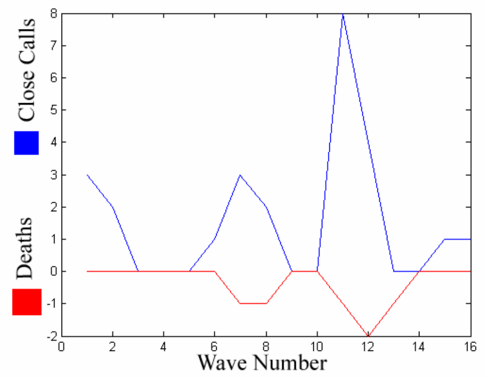
Yet another problem that resulted even with the above modifications was getting stuck at local optima. At local optima, the agent would repeatedly send out identical or near-identical waves of enemies, and receive a small positive reward each time. Although mathematically it is a "good" thing to be consistently receiving positive rewards, this can start to irritate the player, because it fails to provide gameplay variety. This problem was solved by slightly increasing the estimated reward for actions that have not been used recently, because it is possible and likely that the player's abilities have changed over time. As a result, if the same action is used repeatedly, other actions will eventually appear more favorable, and the agent will select a different action.

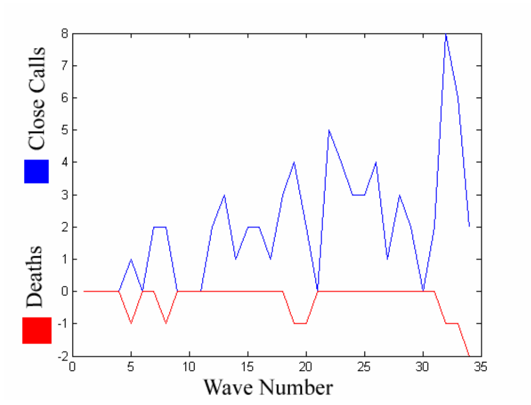
The first implementation of this learning algorithm involved only one wave being sent at the player at a time, with the next wave appearing only after the previous wave was finished. After this was working well, the game still never got quite difficult enough for more skilled player. To fix this problem, the algorithm was extended to handle multiple waves at once. This modification allowed the agent to modify the rate at which it sent waves at the player. With this extension, rewards were distributed equally among all currently active waves. A queue was used to store the action associated with each active wave, as well as the state at which the action was executed. As waves finished, their entries were popped from the queue, and used to update the model and value function as in the original single-wave case.

Experimental Results

In order to judge how well the algorithm was performing in adapting to the player, the number of close calls and deaths for each wave was recorded for several players. An overall high number of close calls, with a drop in close calls after a death would indicate that the algorithm is performing well. In general, a high number of close calls is desirable. But after the player dies, the game should become less challenging, so close calls should drop.

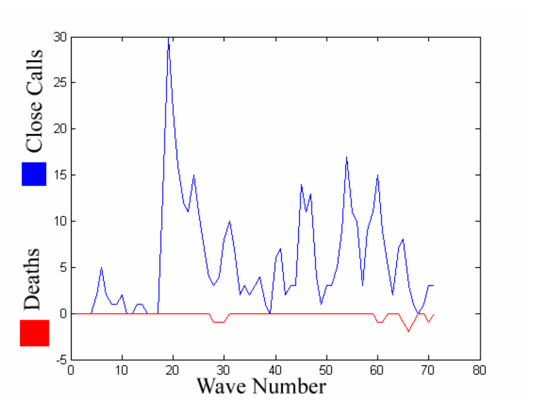
The following plots show the numbers of close calls and deaths for different players. The first plot shows a player who only played the game for 16 waves. The player's deaths occurred during two parts of the game, during each of which the number of close calls was very high, indicating that the player was being challenged. Also, after each death the difficulty dropped, as shown by the decrease in close calls following each death.





This next plot shows a player who played for a somewhat longer time period of 35 waves. Again, each death was shortly followed in a decrease in difficulty. Overall, the number of close calls per wave was high, only dropping to 0 during a small number of waves.

The final plot shows a more skilled player, playing the game for a much longer time period of 72 waves. After about 17 waves of gameplay, the number of close calls remains above zero for all except for two waves. In most later waves, the number of close calls fluctuates around 8. As before, the difficulty tends to drop after each death.



It is important to note that graphs are generally not a great measure of player enjoyment. In addition to collecting numerical data from several players' experience with the game, I had ten people play the final version of the game for as long as they could survive, and give me feedback about how fun they thought the game was. All players reported that they found the game to be both very enjoyable and intense, and a few told me later on that they were addicted. However, some players found that the game got difficult too quickly during their first game, but in their second or third game after knew how to play, they thought that the difficulty adapted well. The overly quick increase in difficulty in the first game was probably caused by the automatic negative reward for each wave being too large in magnitude, causing the game to get difficult a bit too fast. Future testing can be used to find a better value for this reward. Or perhaps another learning algorithm can be used to find better rewards associated with close calls, deaths, and waves over the course of numerous games.

Overall, I believe that this project was a great success, yielding a fun game that is innovative both in terms of gameplay and in its application of machine learning. Countless extensions of this project are possible, including the use of this idea in more complex types of games. For example, a "bot" in a first-person shooter could learn that the player is a "camper" (player who prefers to remain stationary while letting the opponents come to them) and adjust its strategy to account for this. Or, an AI opponent in a real-time strategy game can learn what types of units and attack patterns a particular player has the most difficulty defending against. The ideas I have developed while working on this project could potentially be used to help make games of various genres more enjoyable and adequately challenging for all types of players.