Automatic Character for Pathfinder

Xavier Humberg and Nathan Rogers

***Abstract*— In this paper, we present a basic plan for a machine learning model for training an A.I. to play the Pathfinder Role Playing Game. We will then discuss the results of creating a Fighter in the system and discuss why we received the results we did.**

I. INTRODUCTION

The program we made for this project was an automatic fighter class character for the tabletop roleplaying game Pathfinder. We decided to make an automatic character to fill a role for times there are not enough players to go through a campaign. The idea is that the automatic character would be able to fill a role for a missing player. We decided to start with a fighter class for a few different reasons. The main reason is that there are fewer players that play the fighter class role and would more likely be in need of filling. Another reason is that the fighter class is a little more straight forward with actions to take each turn. The fighter class, in Pathfinder, serves as a kind of tank role, and should be mostly focused on getting up to enemies and attacking in order to keep the enemies attention off the other characters. To this end, our program keeps track of how many enemies there are, how big they are, the distance to the closest enemy and the biggest enemy, how much health the character currently has, the highest amount of damage an enemy has done, and the hit rates for both the biggest enemy and the character. This would also serve as a good starting point for future implementations of other character classes as they would also need this information plus some additional information important for each specific class.

Pathfinder is a tabletop role playing game and is really only limited by the imagination of the dungeon master. We wanted to use machine learning techniques here instead of traditional techniques so that the program can be presented with any scenario and make a prediction for the best course of action. It would be difficult to use traditional techniques for this, as we would need to program each exact situation. With nearly infinite possibilities within even a single campaign, we needed a program that could learn what actions to take with just the information available to players on any given turn. This would allow the program to be used whenever and wherever needed.

II. IMPLEMENTATION

We decided to implement the project with a two-layer system consisting of an initial layer of a decision tree, followed by a layer of perceptron. The decision tree layer reduces our list of choices, and the perceptron layer confirms the choices made by the decision tree.

*This work was not supported by any organization

A. Decision Tree Layer

The first layer uses decision trees to narrow all possible actions down to the three actions that make the most sense with the available information. For example, we dont want the program to try to attack when the enemy is too far to hit. For this layer, we use three decision trees, one for each of the three possible actions. We trained each tree a little differently in order to get three different actions. The first tree was trained as the most aggressive choice, with most of its actions being to charge, or to do a full attack. The second tree was trained with more strategic choices, like flanking, fighting defensively, or just defending if health is low. The third tree was trained with more passive actions, like step away and heal, or even retreating if needed. The three choices would then be fed to the second layer of our program, which would then make a prediction of which would be the best action to take in each scenario.

B. Perceptron Layer

The perceptron layer is responsible for reducing the number of choices from 1 to 3. This was accomplished by ordering the trees and using perceptron on the results obtained. The ordering of the trees would change from character class to character class, but we'd always take the first tree's result if the perceptron net returns 1, the second if the first tree's result was 0 and the second trees was 1, or the third tree's result if neither the first nor the second returned 1.

For example, since the character we implemented was a fighter, we would always consider the aggressive tree's action choice first. If perceptron returned a 0, we tried the middle tree. If the middle tree returned a 0, we chose the passive tree's options. This is because, in Pathfinder, a good offense is usually better than a great defense.

The perceptron nets were trained via the same training data as the decision tree. In the column following a prediction, we rated the prediction based on whether or not it was a good option for this specific data set. The perceptron data was meant to take into account more things than the decision tree, and to also help alleviate the overfitting problem of decision trees. 33

III. TESTING

Our testing consisted of more randomly generated examples which we applied a singular label to. We then ran the data through both of our layers to receive one, singular action. The error was expected to be suprisingly high, due to the fact that all of our data was fabricated and processed by hand. None of our data was measured, but merely created.

Upon testing our data, we consistently score around 25% accuracy. The highest being 33%, and the lowest being 22%. The variance is due to the random ordered application of the Stochastic Gradient within Perceptron.

IV. POSSIBLE FUTURE PLANS

Our future plans include implementing higher levels for the fighter class character, which would include more possible actions to take. We also have plans to implement other class types, which would include different actions and/or different information pertaining to each specific class to make decisions. Our system is already set up to take possible higher class levels, and all we'd have to do is create another data set for the values. Making spellcaster classes would likely require yet another tree full of spell actions, healing actions, or other class specific actions as well, but they would also need to know several more important, key details for spells such as multiple enemy locations so as to detect if a lightning bolt (a line attack) can hit multiple enemies, or to detect whether a fireball (a burst attack) could do the same.