Cody Hertz

Machine Learning Course Project Report

Predicting Outcome of a Quickplay Match in Overwatch Using Neural Networks

Introduction

The program I created uses multiple models to be able to predict the results of a match of Quickplay in Overwatch. Of these models, half of them are different Deep Neural Networks. There is no evidence of this being done by anyone before this, there are similar applications for other games, and sports, but there is no evidence of one being created for Overwatch before. The program itself is written in the Python programming language with the three major libraries being used are Flask, TensorFlow-GPU, and SciKitLearn. Flask is an HTML template rendering library that allows me to make a web application to interact with the models and do view different data involved with the program. TensorFlow is a data flow library that I am using to create, train, and run my Deep Neural Networks. SciKitLearn is being used to create, train, and use the SVM and Logistic Regression models in my program.

The reason for choosing Quickplay as the game mode is due to the usual lopsidedness of the teams. Since it is a noncompetitive game mode, players don't try to win as much and this causes games to be much easier to predict solely off of which heroes are chosen, or the team compositions. This helps increase the accuracy of the models, as well as the time to train, leading to very high accuracy with very low numbers of epochs.

Data and Preprocessing

The training data is a list of lists where each list is in the following format: [outcome(0/1), map(0-18), position(0/1), team member 1-6(each individual value can be 0-29), enemy member 1-6(each induvial value can be 0-29), team tanks(0-(6 - (team dps + team healers))), team dps(0-(6 – (team tanks + team healers))), team healers(0-(6 – (team tanks + team dps)))]. The complex dataset is an untouched version of this, while the simple dataset is the first three elements concatenated to the last six elements of the original dataset. All data in the dataset was recorded by me, as no related datasets exist on Kaggle. The data comes from real matches of Quickplay, excluding games where players lost on purpose. The game matches players based off of skill which ensures that data wasn't skewed by difference in skill levels. To preprocess/augment the code the original list is first read from a csv file and then converts byte name values to integer values in the range stated in the format above. From there, teams, position, outcome, team compositions, and enemy compositions are flipped to create a data point where the enemy is treated as the player's team to give both sides of the match. After that, 65 permutations of your team is concatenated to 65 permutations of the enemy team to teach the model that positon on the team doesn't matter, just the heroes included. Finally, the data is normalized and then randomized the order to ensure that the model doesn't try to always predict a certain outcome. It was then rewritten into a new csv file. There is also a smaller version of the dataset to train the SVMs off of due to the time to train the SVMs not being worth the low accuracy obtained from the models. The end size of each dataset is approximately 1.06 million for the normal set, and 110,000 for the smaller dataset.

Models

My models are divided into two major categories, complex and simple. These categories correspond to the dataset they are trained and designed off of. Complex is the complete dataset with all the information, while the simple category uses a subset of data in the complex dataset. This subset contains only the map, position, team composition, and enemy composition. Composition refers to the number of each class of hero on a team, Ex: 2 2 2 means two tanks, two damage heroes, and two healers. Each category has a seven layer Deep Neural Network, a five layer Deep Neural Network, a Polynomial SVM, and a Logical Regression model where the inverse of regularization strength is 100. The only difference in the complex and simple Neural Networks is that the first layer in the complex networks have 20 nodes and the simple has 8. Besides that, for the seven layer Deep Neural Networks, the node layout from input layer to output layer is (20/8) -> 1000 -> 1000 -> 700 -> 700 -> 500 -> 2. The layout of the five layer Deep Neural Network is (20/8) -> 1000 -> 700 -> 500 -> 2. The last layer uses 2 nodes because I make use of a one-hot array for the output of the network and use argmax to evaluate if the result is a victory or a defeat. All of the seven layer Deep Neural Networks use a batch size of 500 and iterates over 50 epochs, while the five layer networks only require 30 epochs to reach their optimal accuracy to training time ratio. The activation function for all nodes but the output nodes is Rectified Linear Unit activation. The cost function for all networks is the Softmax Cross Entropy function, which just makes sure the sum of all input is equal to 1. The optimization function being used is the Adam Optimizer with a learning rate of 0.001.

Limitations

There are several limitations of the program that I created. The largest limitation is its inability to predict accurately unlikely or unrealistic games. For example, your inputted data for prediction could have all six members of your team, but the enemy team has 0 members, although your team should win, because this would mean all enemy members left the game, but the models would have a very difficult time predicting this because there is no training data to support this. Even after over 2000 hours of playing the game, I have never seen this happening so it would be very unlikely to train a model with the capability of correctly predicting this result without artificially creating data that reflected this. Similar input data to this also have difficulties due to the rarity of their occurrence. In short, accurate prediction of strange input data that would be easily predictable by a human, is impossible without artificial data.

Some limitations are limited only to the simple models. Although they generally have greater levels of accuracy, they have a very difficult time correctly predicting more fair matches. This is due to the exclusion of character data from the training of these models. This problem is mostly solved by the complex models, but this is sometimes difficult to predict for most players.

The accuracy of the SVM and Logistic Regression models is very low, however this is expected and these models are mostly included for the purpose of comparison and were expected to perform poorly. Also, due to the long training time of SVMs, the smaller dataset was used to train the SVMs.

The final limitation is that Flask is a template generating library. This heavily limited my design choices. For example, I had hoped to display and change images of heroes and map dynamically when interacting with the forms. However, integration of Ajax or Socketing would help alleviate this issue, but I have no experience with Ajax, and Flask-Socket would require a complete rewriting of much of the frontend. As interface design was the least important part of developing the application, I felt that these additions would negatively affect the outcome of the project. Due to this, I decided not to dedicate the team to their inclusion.

Application Design

Flask allowed quick development of the frontend of this application. While, I have much more experience with using Flask mostly as an API and use frameworks, such as Angular5, for the frontend and these give me access to dynamic loading, which would fix the aforementioned limitations, this would have required almost double the time required to implement. The home page is used to convey the purpose of the program, as well as act as a navigation page to different pages.

The new data entry page allows users to enter new data that can be later used to retrain the model. On submission, the new data is reformatted into a list of lists, which only includes one list within the outer list. This allows me to use the functions used in the preprocessing of the data to properly augment it and append it to the end of the dataset used to train the models. Augmentation of the new data is required due to the way SciKitLearn's score function works. The limitation this incurs is that the dataset must be an even number. This is because it requires the test data and training data to be the same size.

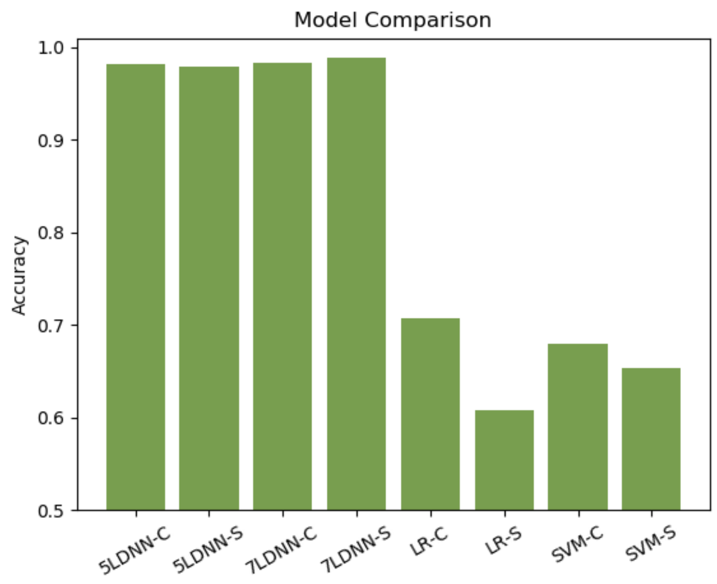
The prediction page is how a user can feed input data into one of the models to predict the outcome of a match. It includes validation of input, so does the new data entry page, to insure valid entry of data. It then uses whichever algorithm was chosen to predict the outcome of the match and displays it a flash message at the top of the screen.

The relearn page's purpose is solely to retrain all of the models. This includes a warning, because of the long waiting time, to dissuade those who do not wish to wait for the completion of the training of the models. If training is stopped part way, the incomplete Neural Networks will be inaccurate, and the uncompleted SVMs and Logistic Regression models will not work. Another precaution to dissuade pressing of the button is the non-inclusion of the link on the navigation bar. When the retraining of the models has completed, the page will load and notify the user of the completion of the training.

The final page is the comparison page. This page displays a graph displaying the accuracies of the models, as well as printed values of the accuracies. Both are included due to the similarity between the multiple Deep Neural Networks.

Results

Below is a graph that displays the different accuracies of the different models:



To better compare the different models here is the percentages display in text:

7LDNNC:	98.28254580497742%
7LDNNS:	98.90792965888977%
5LDNNC:	98.13981652259827%
5LDNNS:	97.86187410354614%
SVMC:	67.93531775474548%
SVMS:	65.27827382087708%
LRC:	70.67604660987854%
LRS:	60.743677616119385%