A.I. Project to Determine Best Winning Practices in Heroes of the Storm

by Alex Laswell, Damian Armijo, and Daryn Butler, December 7, 2018

Introduction

“Heroes of the Storm is a multiplayer online battle arena video game developed and published by Blizzard Entertainment for Microsoft Windows and macOS, which released on June 2, 2015. The game features heroes from Blizzard's franchises including Warcraft, Diablo, StarCraft, The Lost Vikings, and Overwatch. The game uses both free-to-play and freemium models and is supported by micropayments, which can be used to purchase heroes, visual alterations for the heroes in the game, and mounts. Blizzard does not call the game a ‘multiplayer online battle arena’ or an ‘action real-time strategy’ because they feel it is something different with a broader playstyle; they refer to it as an online ‘hero brawler’. Heroes of the Storm revolves around online 5-versus-5 matches, operated through Blizzard's online gaming service Battle.net. Players can choose from different game modes, which include playing against computer-controlled heroes or other players. Initially, no heroes are permanently available for use; however, players may choose from a list of heroes that are free to use from a weekly rotation. By using gold coins, the in-game currency, or through microtransactions, they can gain permanent access to a hero. As of November 2018, there are 80 heroes in the game, divided into four separate roles: Assassin, Warrior, Support, and Specialist. These heroes include one multiclass hero. In addition, there are currently 15 maps available to play, each of which has different objectives to secure, with some having different victory conditions. Experience points, which can be gained by being nearby enemy units when they're killed, are shared across the entire team. When a team reaches a certain experience point threshold, every hero on that team levels up, acquiring slightly amplified powers. Every few levels, players may select a talent which offers a new ability, or augments an existing one. This leveling system emphasizes the importance of teamwork and planning, since a player's action can affect the whole team. Players can also mount different animals, such as horses, lizards, or unicorns, to increase their movement speed, automatically dismounting when attacking, receiving damage or using any ability.” *(Wikipedia)*

Damian and Daryn are on the collegiate Heroes of the Storm team and as such spend a lot of time practicing playing the game and are very interested in improving their play for the March tournament. As such, we decided to do a project to try and maximize our odds of winning in Heroes of the Storm based on a number of previously identified important statistics found within the game. Each player chooses a hero, and throughout the game tries to score takedowns, clear enemy minions and destroy enemy buildings, all while attempting to die as little as possible. As these are the key points of the game, we created a program that would combine these key statistics for a single player across multiple games that they have played and use neural networks to determine what stats are most impactful on the result of the game, as well as some other effects like which maps and heroes the player is best at. The game outputs an encrypted proprietary file called a STORMREPLAY file (.StormReplay) that essentially replays the game, storing every player action taken over the course of the game as well as every stat and end-of-match award. However, the file isn't plaintext and requires a secondary program to decrypt.

Once we decrypted this and parsed it into a data set that we could use, we ran a comprehensive test on the activation functions in our neural network and size of neural networks, and tried to make a decision on which configuration would be the best to use for our testing. We then took this optimal neural network setup and ran some tests as described below.

Methods

Daryn's Task(s)

Daryn was tasked with gathering the data and putting it into a format suitable for running the AI algorithms on. He used an open-source program called Hero Protocol (https://github.com/Blizzard/heroprotocol), a Python library to decode and parse a .StormReplay file, which is how the game saves its replays. The program breaks up a replay into 7 major sections (*initData*, *Header*, *Messages*, *Details*, *Attributes*, *Tracking Events*, *Game Events*). Each section is accessed via a command ("python heroprotocol.py") as well as optional flags (--json if you want the output to be JSON formatted, and "--initdata", "--header", "--messages", etc. for which large section you want to view). Most of the sections aren't relevant to our study. *initData* contains the sync data of the lobby that the players join to start the game. *Header* has mainly client information like the version, security signatures, build info, and public keys. *Messages* contains any messages or map pings players have sent to each other. *Attributes* is mainly metadata, and *Game Events* contains all of the player's movement commands. Our main points of interest are in stats and pregame decision making. These are contained in *Tracking Events* and *Details* respectively. *Details* contain the map, the player's chosen character, as well as cosmetic choices like their mount and character tint. It's also one of the places the end result (win or loss) can be found, and we found it easier to determine the result there than in *Tracking Events*. This is because *Details* contains, among other things, a data structure called "m_playerList". This structure contains 10 entries, one for each player. Each entry is a dictionary which contains general information about each player including what team they are on, their chosen character, their username, their tint, whether they are a player or observer, their server, and whether they won or lost. That means this list provides some of the stats we wish to use in our A.I. program quite readily. Pulling the other statistics was more difficult because the *Tracking Events* system is massive and inscrutable, containing, among other things, quintuple nested data structures. There are 12 event types contained in *Tracking Events*, and the relevant event for us was event 11, the score screen results. Through testing we determined we could get to the stats like so:

Tracking Events >>> dictionary
Event 11 >>> list
All Stats >>> dictionary
Specific Stats >>> dictionary
Player’s Stat >>> one-entry list containing a dictionary
This was the trickiest part of parsing the data and took some tweaking to finally get the data how I wanted it. My goal was to have a list of games, with each entry being a list of all the player’s relevant stats we wanted to look at. We used Damian as our test subject, being sure to only pull out the stats of username “silkylaroux”. This in itself posed a challenge as the username is contained in the *Details* section, but not the *Tracking Events* section. Through some experimentation (as well as referring to the match history in-game) we determined that the index of the player in *Details* corresponded to the index of the player’s stats in Event 11. Once we had that, we added the index to the info we pulled from *Details* and then it was much quicker to pull stats from *Tracking Events* as the dictionary of a specific statistic had two entries: “m_name” which was the name of the stats and “m_values” which contained that statistic for each player. The list that served as the key in “m_values” was the list that matched the indexing of the players in *Details*. Each entry in that list was a one-item list containing a two-item dictionary, with the entry “m_value” being the stat we wanted (it also contained the game’s runtime in “m_time”). We now had all the stats that we felt were necessary: the player’s character, the map they played, their takedowns, deaths, damage to heroes, minions, and structures, and mercenary captures. This was formatted in easily accessible in lists of lists by way of the following code that I wrote, separate from the code in the report:

```
gameData = []
gameNames = []
directory = 'C:\\Users\\LichKing\\Documents\\Jupyter Notebooks\\440Project'
for filename in os.listdir(directory):
        if filename.endswith(".StormReplay"):
        gameNames.append(filename)
        continue
        else:
        continue

gameNames

count = 0
for name in gameNames:
        stats = []
        details = ripData(name, True, "--details")
        if not details[0].__contains__('Unsupported base build:'):
        details = json.loads(details[0])
        position = 0
        for x in details.get('m_playerList'):
```

```
        if x.get('m_name') == 'silkylaroux':
                if x.get('m_result') == 1:
                stats.append('WIN')
                else:
                stats.append('LOSS')
                stats.append(details.get('m_title'))
                #print("Game",count,":",x.get('m_name'),"is playing",x.get('m_hero'))
                stats.append(x.get('m_hero'))
                trackEvents = ripData(name, True, "--trackerevents")
                trackEvents = getMessages(trackEvents)
                data = []
                for y in trackEvents:
                if y.get('_eventid') == 11:
                data.append(y)
                data = data[0]
                data = data.get('m_instanceList')
                for z in data:
                if z.get('m_name') == 'Takedowns':
                stats.append(z.get('m_values')[position][0].get('m_value'))
                if z.get('m_name') == 'Deaths':
                stats.append(z.get('m_values')[position][0].get('m_value'))
                if z.get('m_name') == 'HeroDamage':
                stats.append(z.get('m_values')[position][0].get('m_value'))
                if z.get('m_name') == 'SiegeDamage':
                stats.append(z.get('m_values')[position][0].get('m_value'))
                if z.get('m_name') == 'MinionDamage':
                stats.append(z.get('m_values')[position][0].get('m_value'))
                if z.get('m_name') == 'MercCampCaptures':
                stats.append(z.get('m_values')[position][0].get('m_value'))
                count = count + 1
        position = position + 1

        if stats:
        gameData.append(stats)

gameData
```

The 2D list gameData can then be used in the neural network code to create predictions of a target field (in this case, winning or losing).

Alex's Task(s)

Alex mostly focused on getting the neural network code setup and ready to run for any activation function that we could want to utilize with the data set that Daryn created. Alex set up a neural network class that takes the input and target fields as lists (as we have done all semester long), and handles all of the steps necessary to feed them through a neural network. This means the code standardizes the data, defines the activation function and its derivative, and provides a train and use method to allow us to test any data set, just like we did for our coding assignment five in this class. This is the NeuralNetwork class that is defined in the code. Once the initial NeuralNetwork python code was set up, it was really easy for Alex to then abstract out the activation function and the activation derivative function, so that we could define different activation functions than tanh and compare and contrast the results. This is the NeuralNetwokReLU class that is defined in the code. The team had initially planned on further defining the swish activation function and adding the results from this activation function to the result set for comparison, however we unfortunately ran out of time and were not able to complete this.

Damian's Task(s)

The main task that Damian took up was that of dealing with the results that the different Neural Networks produced. Once the code was running for actually parsing data, and then training a neural networks with this data testing needs to been done to assess the usefulness of the data gathered. There were a few different things that should be addressed when looking at how to test the data. We needed to make sure that we had the correct partitioning of data into the proper input(X) and target(T) sets. We used the game data which was ripped into a txt file, and partitioned the data into the X inputs, which were: ['Map','Character','Kills','Deaths','Hero dmg','Siege dmg','Minion dmg','Merc captures'], and T targets which were: ['Win','Loss'].

This was done via three separate functions, the first being, readDataFromFile(file). It made use of the python module Abstract syntax tree (ast) and evaluated the already formatted data in the file, and put it into a single list, which is turned into a numpy array. The next function changeToInt(nparr,col), goes through the particular column in the numpy array given to it, and returns a dictionary where each unique (non Integer) value in that column is mapped to a unique integer. These dictionaries then get put in the function replace_with_dict(nparr, dict) which replaces each of the non-Integer values with their corresponding integer. This array which is now formatted correctly was then split up into the X and T numpy arrays.

Once they were split we partitioned the data into testing and training data using the partition(X, T, percent, shuffle) function. Which separated the values into testing and training arrays which we can use on the created neural networks. We now had all that we need to effectively test the data, so we did some initial testing with some randomly chosen hidden layers, at 1000 iterations. It was nice because we were testing neural networks but, we were able to test 2 different activation functions. We ran some tests on both the ReLU and tanh activation functions, and printed out from the neural networks draw to get an idea of what inputs seemed to be effected the most. After doing this I printed out a plot mapping out how well the neural network did when used on some testing and training data, since there was only two target values it was easy to see if the neural network clumped around 1(Loss) or 0(Win) or if it was unsuccessful at learning anything. After doing this for both of the activation functions, I made a neural network with the NeuralNetworkClassifier class so that I can print from the confusionMatrix(Ttest,Ptest,classes) function in mlutils. This prints out the confusion matrix, which shows how often the neural networks incorrectly guess a win or a loss.

After doing all of these things we moved on to run tests on a number of many different hidden layers, and see what the best hidden layer is. We check this by finding out what the rmse from each of the different hidden layers that we ran. After finding which layer has the lowest rmse we then ran it with more and less iterations, to see what the best “options” for the neural networks are. We then made plots for the “best” neural networks to show how effective overall we were able to determine wins and losses by using our neural networks.

There were a few reasons why we choose to do the testing that we did. We decided that it was necessary to look at multiple neural networks with different factors being changed. This was so that it could give us a better way to judge if our neural networks were actually performing better or worse. We then were able also use the “best” neural networks that we found to try and get the most accurate results that we could. It also can help show some trends in the data, like what input may seem to have the highest or lowest impact on determining the output. This way of testing could also help in informing later testing on what to add or remove in terms of inputs. Overall there was a good amount of testing done on the different activation functions, and the results will be discussed further below.

Finally, as a team, we all met every week, sometimes multiple times in a week, and we all actively participated in every aspect of this project. While each member did have individual tasks we completed, more often than not the other two were in the room or on discord helping with each thing. This project definitely would not have been as comprehensive or complete without all the team work and dedication of each member, and we are all extremely happy with how everything turned out.

Results

All of our code for this project can be found in the NeuralNetworks.ipynb attached to this report. The notebook outlines exactly how we ran each cell and has some really detailed graphs along with some comments about our findings. However, we have also summarized the results below per the instructions.

We found some very interesting results from running our neural networks with the two different activation functions. One of the first things, which we found by looking at the weights of each of the different with different hidden layers, had to do with the inputs:

INPUT INFO FROM WEIGHTS

- Map:
  - Overall it seemed like the maps didn't have a large impact on determining if it was a win or not. The magnitude of the boxes were generally consistent and didn't seem to be either positive or negative.
  - It seemed also to be pretty consistent between the RelU and tanh activation functions.
  - This would make sense in terms of the game, because the maps overall do not generally allow a win or a loss over other maps.
- Character:
  - It seemed like the character that was played overall didn't seem to determine a win or a loss. It did seem though that it did have large impact on some of the specific layers. A few of the layers had large magnitudes, both positive and negative.
  - This result could make sense because Damian would be generally decent at most of the characters, but a few would be either really strong or really weak.
  - It could also be that the some characters weren't played very often and thus the magnitude is greatly increased just by there not being many data points.
- Kills:
  - This was the first input which seemed to have something definitive it was interesting though that it was a negative.
  - It had a very large impact on most of the different layers, and almost all were negative.
  - We were unsure of why this would be the case, in terms of the game, and couldn't think of why the amount of other players killed would result in only a negative weight.
- Deaths:
  - This seemed to be exactly opposite from Kills in terms of being weighed negatively. It seemed the amount of times that 'silkylaroux' died did have a large impact on the neural network, and it was weighed very positively.
  - We couldn't decide on why it would be impacted positively, but we do think that the more deaths in games would have a large impact on if the game was a win or a loss.
- Hero Damage
  - This is the damage that was done to other players in the game, and we predicted that it may play a large factor in the neural network. It ended up being fairly mixed results however, the magnitudes overall were not too large compared to the other inputs, but a few (both negative and positive) were pretty large.
  - It seemed like it did have an impact, but not as much as we had thought. It also was unclear on if it would be a good or a bad metric for the neural network.
- Siege Damage

    - It does seem to have a large a large impact on the neural network, but it was split on if it was positive or negative.
    - Siege damage with the relu activation function seemed to have a fairly large impact, and it was positive for the neural network.
    - The tanh activation function also seemed to be affected by Siege damage, but it was negative.
    - We couldn't speak to why this was the case, because it was so split between the activation functions.
- Minion Damage
    - This seemed to have less of an impact on the magnitudes, it also didn't seem to sway one way or another.
    - This could be because this is generally pretty uniform, and damage to minions is consistent throughout many of the games.
    - It could also have less of an impact, because killing minions may actually not have much of a contribution to the outcome of a game.
- Siege camps
    - This did seem to not have a very large impact on the magnitudes, but it did seem that it tended to be a positive for each of the different activation functions.
    - This would make some sense, because some maps and characters do not have much to do with camps, but it is almost always good the more camps that are got in a game.

OVERALL Neural Network SUCCESS

When looking at the success of our neural networks we made scatter plots which show the actual outcome of a game vs what our neural network predicted for a game. This was pretty interesting because we could actually see that the training did make a difference in telling if we were going to win a game or lose it. For the relu activation we could see that it did tend to clump around either a win or a loss after it was trained. The test data seemed to be completely random, and didn't clump towards either a win or a loss. The tanh function seemed to work even better with the neural network, and it clumped even more, with less random predictions in the middle. It was interesting to see though that the neural network tended to predict losses more than it did wins. Something that would be interesting to see, is if more of an emphasis on the larger inputs would make a the neural network tend towards wins or not.

Another smaller thing that we used to judge the overall success of our neural network is by printing a confusion matrix. This shows how often the neural network was correct in guessing a win or a loss. It ended up guessing correctly 79.7% of the time for a win, and 76.7% of the time for a loss on the trained neural network. This is pretty good, it showed that it did actually do a good job at guessing. The machine learning utilities also gave a percent correct for training and testing, it was 99.71% for training and 78.65% for testing. Overall the neural networks did seem to make some really interesting conclusions, and it also seemed that it was able to at least somewhat accurately see if a game was a win or a loss.

Conclusion

Daryn's Conclusions

To conclude I'd like to talk about some of the challenges I faced when working on the data. As mentioned before the hero protocol library is massive and somewhat confusing. Damian and I spent a good amount of time just messing around with the code, trying to find what information was stored where and how to access it. We discussed potentially working through the *Attributes* section but deemed it too challenging to parse through (its entries all looked like the following):

```
{'3008': [{'attrid': 3008, 'namespace': 999, 'value': 'Obs'}],
 '4035': [{'attrid': 4035, 'namespace': 999, 'value': 'ATYR'}],
 '4036': [{'attrid': 4036, 'namespace': 999, 'value': 'met'}],
```

Some of the values were obvious (like 'value':'mele' being a melee character) but most were not. Parsing the Event 11 data structure was also tedious and took a lot of trial-and-error. It was also quite time consuming as the method for data extraction pulled an entire section from the replay (you couldn't request only Event 11) so that had to be redone for each replay. There was also frequent checking with the in-game match history to confirm that the stats we were pulling matched what we expected. Finally, I'd like to talk a bit about why many statistics got left out. The replays hold an exhausted amount of stats, from damage taken from other heroes, to number of health potions picked up, to number of objectives captured, as well as map statistics like number of minions spawned. I decided many of the specific hero statistics wouldn't do well for predicting stats because different characters are expected to do different things in order to secure victory. For example, judging a healer character for damage taken is irrelevant: the role of a healer is to heal, and the role of a tank is to take damage, so it would be equally pointless to judge tanks by their healing numbers. Likewise, as healing potions often appear in the front lines of a battle, asking a mage, who is easy to kill and designed to stay on the back line of a fight, how many potions they picked up, wouldn't exactly be fair either. And the map statistics like minion spawns are also irrelevant as they are independent of the player's decisions. The stats I chose to focus on were the most generally relevant across all characters, and have major impact on the player's chances of winning. One final note about the use of the hero protocol library was its extreme pickiness with file location. The program uses protocols (.py files) that are applied to replays to format them for data extraction. Older replays are formatted differently than newer ones and require different protocols. These protocols must be kept in the same folder as the main code, and there are a lot of protocols. This problem is exacerbated by the fact that the replays ALSO need to be kept in the same folder as the main code, meaning that trying to navigate our program's structure is time consuming. I would have liked to modify the program so that we could separate these out and create a more organized system but that was out of our time constraints. I already had to push back my original time table as I was far busier during break than I anticipated so I had to rush my data gathering and didn't get quite the number of replays (or the familiarity with hero protocol) that I was hoping. I'm still confident we have a good data set, however.

Alex's Conclusion:

For my formal "part" of the project, I was tasked with setting up the neural network and integrating it with our testing data set.

Overall I would say that this went really well. I have done some previous work in machine learning with neural networks, among other things, so I had a pretty good understanding of what was going to be needed already. Additionally, I had a comprehensive working neural network package that I was able to leverage for this project. That code came from a previous class with Dr. Anderson, which was especially helpful because it allowed everything to really fit together nicely. It definitely still took me a significant amount of time to work out all of the bugs and make sure everything was working correctly, but again overall this was not too difficult and honestly that is my favorite part of coding so it was not bad at all.

Formatting our data to be a list in the right way was also not too difficult as this is how Dr. Anderson has set everything up all semester long and so it was a pretty natural transition. We did need to write out several helper functions to get everything correct, but that was something we all expected and was planned for. Also, as Daryn outlined above, parsing the initial data set into something more manageable and targeted was a task, but once they handed off to me, getting it to feed into the neural network code and everything went well.

Some things that I think that went bad for this project are the just the general scope of everything was too large. Initially, we thought that we were going to be able to not only test all of the activation functions and come up with the sort of, optimal neural network, but then we wanted to really go deep into the testing and try to produce some findings that would represent the optimal choice tree (path) and be able to speak to maps that were best or characters that were going to be the winning choice. However, that ended up being way more than we could actually deliver on. It took us meeting several times a week and really working hard to just get all the data setup, the neural network code going, to write the tests, and interpret the results for which activation function was the optimal and what size of a neural network to work with. In fact, we actually had to make a team decision to stop digging in that direction too, as we all still felt that we could have done some more research on the size, and there were still more activation functions we wanted to test. Unfortunately, we simply ran out of time. We are all planning on doing further work on this as this is something that is a passion for all of us, and we are still really interested in seeing if we can come up with answers like; what is the best player or map or team build to choose to win most often, if one exists, or can we add in the potions and healing to the data and how will that change compare with the results we see. However, in the scope of this project, we simply were not able to get to all of the things we wanted to do in just one semester.

Another thing I think went bad in this project was the holiday break. The holiday coming up in the middle of the semester never helps, but we all kind of forgot to plan for this, so it really threw a wrench into development. None of us were in town at the same time, and working remotely was really difficult, so essentially nothing got done and we had to push our whole timeline back a week and try to catch back up from there. This could have been avoided had we taken the time to look over the semester calendar, and will be something we all look out for in the future.

What I learned in doing all of this is that even though everything sounds so complex, artificial learning with neural networks, it really is not all that hard to setup and run my own experiments. I can read and interpret my results, and with the jupyter notebook, I can easily share my results with the world. I also learned that I can follow another projects API and that I do not need to be told how to do every little thing for my code to work out. Rather having the freedom to choose and the ability to utilize whatever tools I deem necessary was very liberating and has really energized me to delve even further into the machine learning and artificial intelligence world. I am already planning ways of incorporating neural networks into my work at HPE, as well as some personal projects I have been working on, and I have joined a couple of communities that are centralized around machine learning and neural networks as a direct result of our work here.

Damian's conclusion

I thought that it was pretty interesting to see how we can apply the neural network concepts that we learned in class to real world data, and learn interesting things from it. When doing the testing I did have to redo the partitioning of the data, because initially it was not including one of the inputs that was in the data. This was the only real major issue that I had, as I found out about it at the final stages of testing. I also think it was interesting to see that the neural networks performed pretty well, and how the inputs can play different roles in impacting the data. The best hidden layer being run with a larger iteration size also did seem to produce more accurate results, which I thought was cool, because it showed that there is a correct way to train the network.

I would like to continue research on this however, because it would be interesting to see the neural networks predictions with a lot more inputs. It would also be interesting to see if it could find more information on the entire team, and what is most likely as a team to produce a win. I think it could also be applied to more things than just wins and losses, so I will continue doing research in the future. Overall I am really pleased with what we did, and with the results from running our neural network with data that I can use to maybe improve my play in Heroes of the storm.

References

http://www.cs.colostate.edu/~anderson/wp/
(Dr. Chuck Anderson provided the mlutilities code and the neural network code we utilized)

https://github.com/Blizzard/heroprotocol
(The home of the hero protocol library we used to decrypt and parse .StormReplays)

https://en.wikipedia.org/wiki/Heroes_of_the_Storm
(The Heroes of the Storm description we used in the intro)

https://www.hotslogs.com/Player/MatchHistory?PlayerID=7442886
(The match history site to match the stats we were pulling to the actual game history)