Predicting Professional CSGO Player Performance using Machine Learning Techniques

CSCI 4155

James Wilcox - B00743927

1. Introduction

Counter-Strike: Global Offensive (CSGO) is a multiplayer first-person shooter game developed by Valve. From Wikipedia: “the game pits two teams against each other: Terrorists and Counter-Terrorists. Both sides are tasked with eliminating the other while also completing separate objectives. The Terrorists must plant the bomb while the Counter-Terrorists must either prevent the bomb from being planted, or if that fails, defuse the bomb”. In addition to being played by millions of gamers all over the world, CSGO is also one of the largest esports in the world. Over the game’s nine-year history thousands of matches have been played between the best teams in the world for huge prize pools.

While reading “NBA Game Predictions based on Player Chemistry”, a paper on a machine learning project from the Stanford intro course project page, I was inspired to create a similar project with CSGO. Like traditional sports, CSGO esports has seen its share of legendary rosters, heated rivalries, and drama. How players are affected by leaving/joining certain rosters, and chemistry between players is often discussed. I wanted to know if player chemistry in CSGO is mostly just superstition or if it can be quantified and predicted. My end goal is to create a model that can look at both rosters in a game and be able to predict if players will underperform or overperform. My target variable is HLTV.org’s “rating” statistic, which has been designed by CSGO experts to show how well a player performed in a game. HLTV rating has an average of 1.0 and getting a rating of 1.4 would be a superstar performance while getting a rating of 0.6 would be an abysmal performance.

This project is more of an implementation project and was a huge learning experience so I am writing an ad-hoc style report on my experience.

2. Data Acquisition

The first challenge of this project was to acquire a dataset. I needed lots of information about professional CSGO games and there was no ready-made dataset available on the internet. Fortunately, there’s a website called HLTV.org that contains a database of thousands of professional CSGO games going back to 2013. However, there is no official API to download data from this dataset.

I found an “unofficial HTLV API” on GitHub for JavaScript. Using this tool, I started working on a JavaScript Program that would download the data I needed from HLTV. Writing this program was a learn-as-you go experience as I had not written much JavaScript before and I had never created a program that would automatically make thousands of requests to a server. One problem I faced was that, because the API was unofficial, if my program made too many requests too quickly, my IP would be banned from making anymore requests for a timeout period. I got around this by waiting between requests for a time of 3.5 seconds (I experimented to find this particular time), however this caused my program to take over 12 hours to download a few hundred megabytes of data.

Finally, I had data from about 8000 professional CSGO games downloaded into a json file. These were games played between teams ranked top 30 in the world, from 2018 until March 2021. Now I had to extract that data in a way that was usable for machine learning.

2.1 Data representation

My model was only interested in two things: the players in a game, and the rating that each player would achieve in that game. So, my model would take as input the identities of the five players on each team and output a vector of ten ratings, each one corresponding to each player from the input. To represent the identities of the players I used one-hot encoding. Since there were about 500 players that appeared in the dataset, this meant that my input was a 1000-length vector, a 500-length one-hot vector for each team stacked on top of each other. My first idea for representing the output was a 10-length vector of ratings. I thought that each rating could be assigned to a player by assuming the first rating in the output corresponded to the first player that was encoded in the input. This turned out to be a bit of a mistake, as I will elaborate on later.

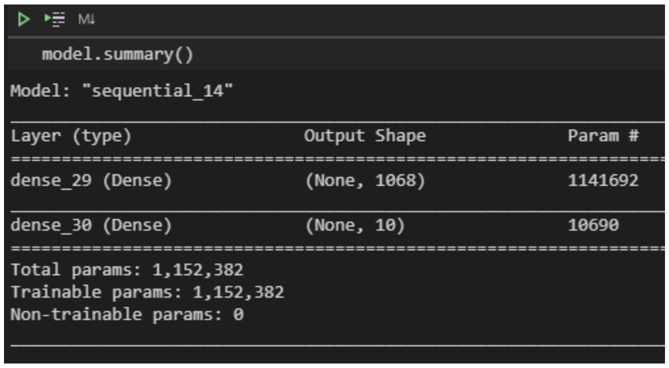
I implemented this with a python script that used the json library to extract the data from the json file, encode it as described above, and save the results in a pandas dataframe.

2.2 Data Augmentation

I augmented the data by “flipping” the teams in each game, so for every game in the dataset I get two training examples, like “team 1 vs team 2” and “team 2 vs team 1”. This amounts to changing how the order of the stacked vectors.

3. First Output Type

My first attempt at training a model on this data used a sequential neural network using TensorFlow. I somewhat arbitrarily chose a hidden layer of 1068 units using relu activation. I chose 1068 because that was the size of my input. I only trained the model for four epochs with a large batch size it would overtrain quite quickly. Using mean absolute error as a loss function over the 10 outputs, the model had a test loss of 0.20, meaning that on average, the predicted rating would be off by about 0.2.



```
model.summary()

Model: "sequential_14"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_29 (Dense)             (None, 1068)              1141692
_________________________________________________________________
dense_30 (Dense)             (None, 10)                10690
=================================================================
Total params: 1,152,382
Trainable params: 1,152,382
Non-trainable params: 0
_________________________________________________________________
```

Figure 1: summary of model used for first output type.

3.1 Why the first output type didn’t really work.

Something I noticed when getting this model to make predictions on unseen examples, is that for each team, it always predicted the ratings in descending order. In other words, the first 5 ratings in the output vector would be suspiciously sorted from largest to smallest, and then next 5 would also be sorted the same way. I realized this was for two reasons:

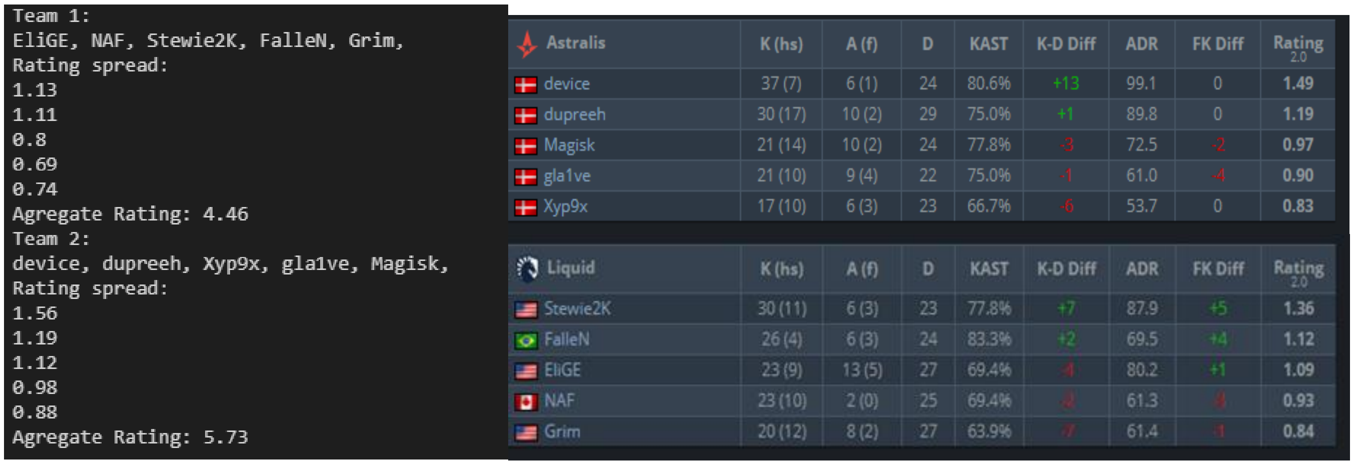
1. The model actually had no way to understand that each output was supposed to correspond to a specific player.

2. The dataset had each game's ratings sorted this way.

So my model was not actually predicting the ratings of each player, it was predicting the *spread* of ratings for the team given a list of five players that made up the team. In other words, it could somewhat accurately predict the right ratings for each team in each game, it just had no idea which player would get which ratings.

Even though it wasn't what I was aiming for, I tested this model by creating a function where I could input a list of players to make up two teams and then get the model to predict the ratings for each team. Some interesting results from these tests:

1. I tested the model on two teams that played a real game lately that the model had not seen, and the results were fairly reasonable.



```
Team 1:
EliGE, NAF, Stewie2K, FalleN, Grim,
Rating spread:
1.13
1.11
0.8
0.69
0.74
Agregate Rating: 4.46
Team 2:
device, dupreeh, Xyp9x, gla1ve, Magisk,
Rating spread:
1.56
1.19
1.12
0.98
0.88
Agregate Rating: 5.73
```

Astralis	K (hs)	A (f)	D	KAST	K-D Diff	ADR	FK Diff	Rating 2.0
device	37 (7)	6 (1)	24	80.6%	+13	99.1	0	1.49
dupreeh	30 (17)	10 (2)	29	75.0%	+1	89.8	0	1.19
Magisk	21 (14)	10 (2)	24	77.8%	-3	72.5	-2	0.97
gla1ve	21 (10)	9 (4)	22	75.0%	-1	61.0	-4	0.90
Xyp9x	17 (10)	6 (3)	23	66.7%	-6	53.7	0	0.83

Liquid	K (hs)	A (f)	D	KAST	K-D Diff	ADR	FK Diff	Rating 2.0
Stewie2K	30 (11)	6 (3)	23	77.8%	+7	87.9	+5	1.36
FalleN	26 (4)	6 (3)	24	83.3%	+2	69.5	+4	1.12
EliGE	23 (9)	13 (5)	27	69.4%	-4	80.2	+1	1.09
NAF	23 (10)	2 (0)	25	69.4%	-2	61.3	-4	0.93
Grim	20 (12)	8 (2)	27	63.9%	-7	61.4	-1	0.84

Figure 2: Prediction from first model

2. When testing a team of extremely high rated players, the best in the world, that had never played with each other against a team called Astralis, which has consistently been the best team in the world for a few years now, the model predicted better ratings for the Astralis players. This may imply that the model is properly capturing the chemistry that these players should have.

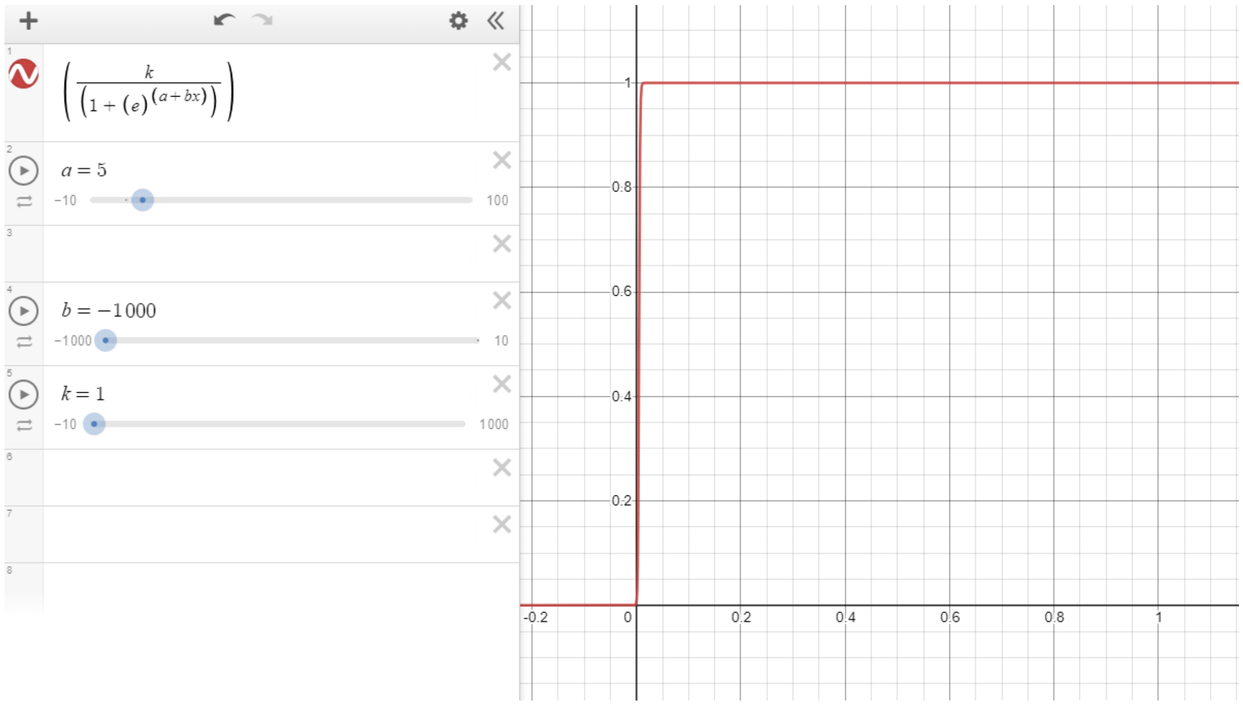
So, this model was not completely useless but was unable to achieve my goal of predicting the ratings of individual players. I probably could have trained it to predict the spread of rankings more accurately by experimenting with hyperparameters, but I instead moved on to solve the problem of predicting the ranking for each player.

4. Second Output Type

My solution to the output problem was to change the output from a 10-length vector to a ~1000-length vector, the same length as my input. Each rating in this vector would correspond to a player the same way the one-hot encoding corresponded to a player.

The first glaring issue with this data representation was computing the loss. In this 1000-length vector there are only 10 values we actually care about getting right, and the position of these values will be different for every different example. If we compute the loss with mean square error the way we did with the first output type, our model will learn to just set all of the outputs to zero since the ~990 outputs that don't correspond to a player in the game are set to zero. This will result in very low loss and the gradient for improving upon this behavior will be extremely gentle.

To solve this, I had to implement my own loss function. I needed to find a way to set the error to zero at all positions in the output vector where the true value is zero, and keep the error as normal where the true value is anything except zero. This could be done by multiplying the error by zero in the former case, and one in the latter. To do this I tweaked a sigmoid function to be very close to the [0,1] step function using desmos. I used the true value as a the input to that sigmoid function, and multiplied the output of that sigmoid function with the error. This resulted in the desired behaviour. I learned a lot about TensorFlow in the process of doing this.



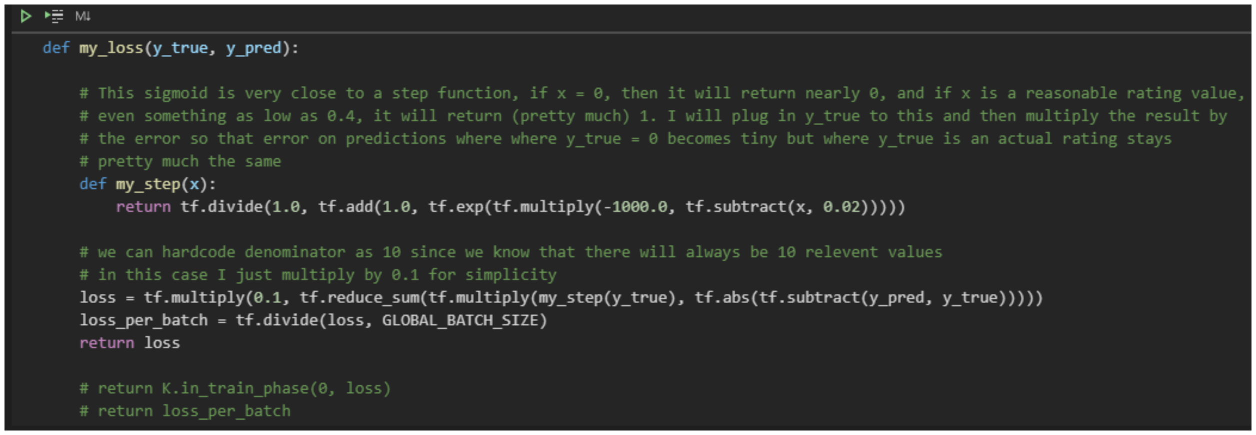
$$\left(\frac{k}{\left(1+(e)^{(a+bx)}\right)}\right)$$

$a = 5$

$b = -1000$

$k = 1$

Figure 3: The sigmoid that I applied to my loss function, taking y_true as an input



```
def my_loss(y_true, y_pred):

    # This sigmoid is very close to a step function, if x = 0, then it will return nearly 0, and if x is a reasonable rating value,
    # even something as low as 0.4, it will return (pretty much) 1. I will plug in y_true to this and then multiply the result by
    # the error so that error on predictions where where y_true = 0 becomes tiny but where y_true is an actual rating stays
    # pretty much the same
    def my_step(x):
        return tf.divide(1.0, tf.add(1.0, tf.exp(tf.multiply(-1000.0, tf.subtract(x, 0.02)))))

    # we can hardcode denominator as 10 since we know that there will always be 10 relevent values
    # in this case I just multiply by 0.1 for simplicity
    loss = tf.multiply(0.1, tf.reduce_sum(tf.multiply(my_step(y_true), tf.abs(tf.subtract(y_pred, y_true)))))
    loss_per_batch = tf.divide(loss, GLOBAL_BATCH_SIZE)
    return loss

    # return K.in_train_phase(0, loss)
    # return loss_per_batch
```

Figure 4: Implementation of Mean Absolute Error with the sigmoid from Fig. 3

Now I can train models with output that correctly corresponds to each player in the input.

4.1 Testing out different Neural Network architectures

To test the model with different architectures I created a compile_and_fit function so I could define different models and pass them to this function. I added a learning rate scheduler to reduce the learning rate as training went on, and early stopping.

At first I tried a simple multi-output linear perceptron, since I suspected that the data might be too sparse to do much better than linear regression prediction. This model had a loss of 0.27. I then added a tanh activation to this model, which improved the loss to 0.25. I then tried some small models with added l2 regularization, testing regularization values of 0.01 and 0.001 (the latter performed better). Finally I tested out some medium sized models with and without regularization and dropout. The multilayer models performed better than the perceptrons, suggesting that the data had at least some more complex and interesting relationships that were learnable.

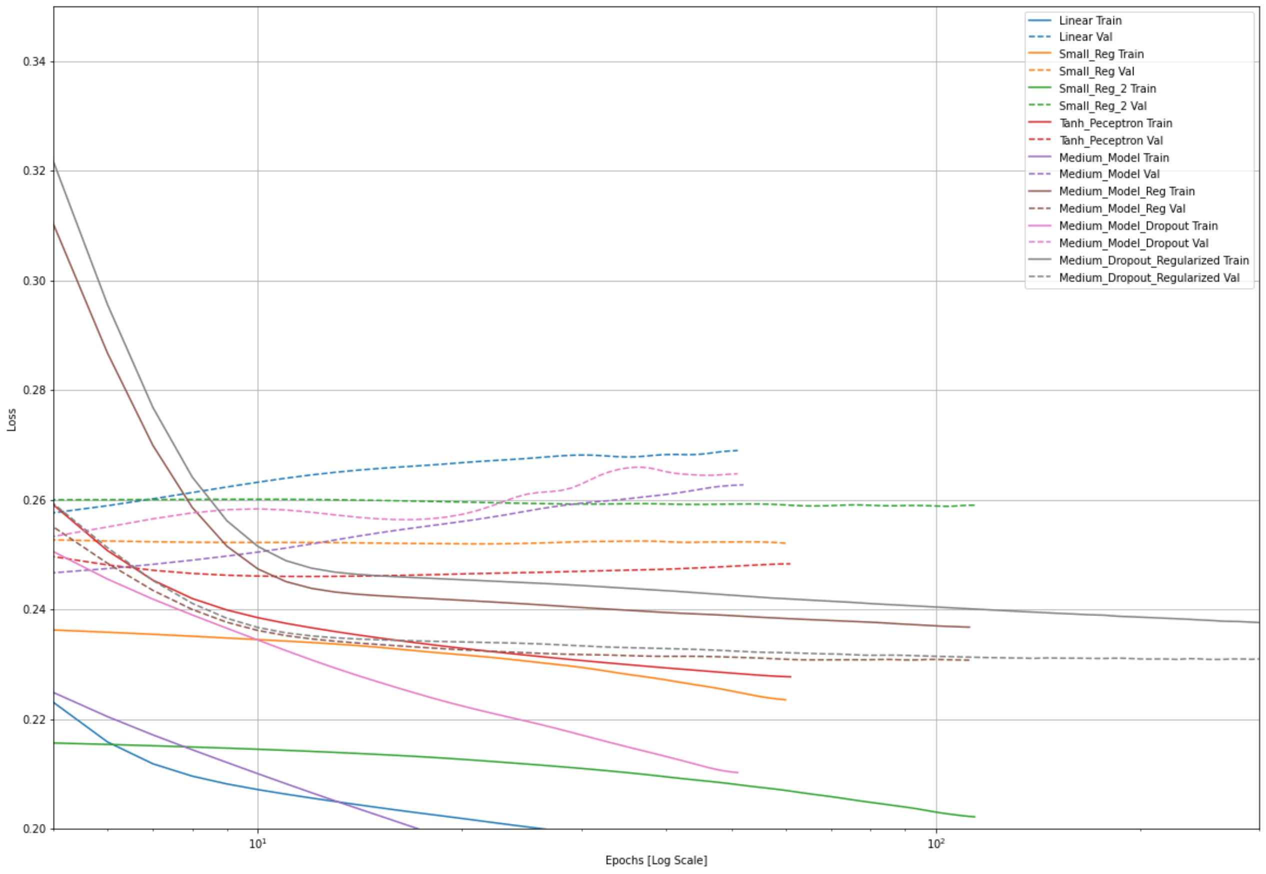
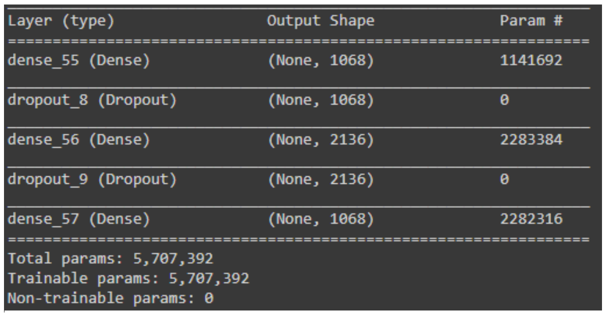


Figure 5: Loss metrics of different model architectures

Since the medium sized network performed best, I decided to try out some larger networks too. Dropout did not help the loss of these medium networks, but I suspected it may still help with larger networks.



```
Layer (type)                Output Shape              Param #
=================================================================
dense_55 (Dense)            (None, 1068)              1141692

dropout_8 (Dropout)         (None, 1068)              0

dense_56 (Dense)            (None, 2136)              2283384

dropout_9 (Dropout)         (None, 2136)              0

dense_57 (Dense)            (None, 1068)              2282316
=================================================================
Total params: 5,707,392
Trainable params: 5,707,392
Non-trainable params: 0
```

Figure 6: Summary of medium model with regularization

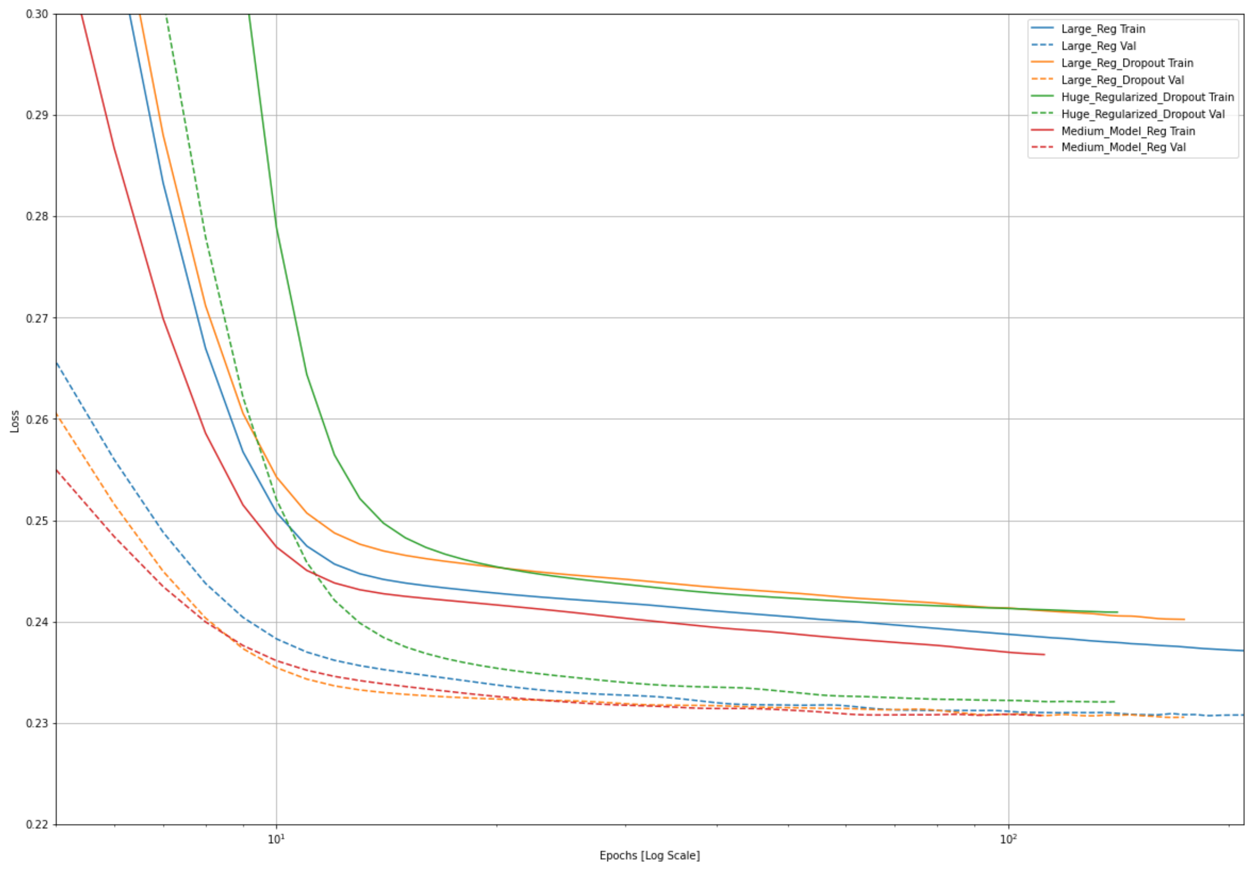
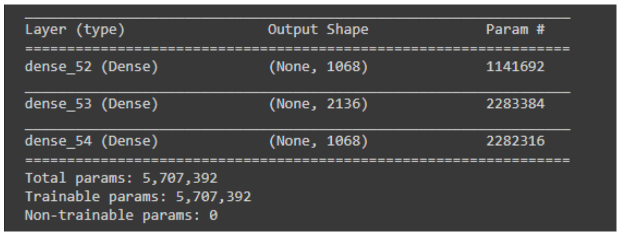


Figure 7: Larger networks loss metric

For the larger networks, my “huge” model, which had over 85 million parameters, did not perform any better than the smaller “large” models. An interesting result is at the large model level, dropout helped slightly, even though it did not help at the medium level. The large models were nearly as good as the medium model from the first batch of architectures, but not quite. I think this implies that this regression problem is only solvable with neural networks to this level of error (about 0.233 on the test set).

Model	**No. Parameters**	**Test loss**
Medium with L2 regularization	5,707,392	0.2331
Large with L2 regularization + dropout	6,849,084	0.2332
Large with L2 regularization	6,849,084	0.2334
Huge with L2 regularization + dropout	85,434,760	0.2345
Small with L2 regularization	2,283,384	0.2521

As a result of these tests, I decided to use the medium model with L2 regularization to play around with the model a bit.

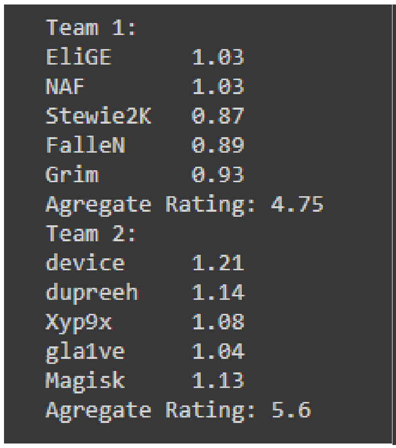


```
Layer (type)                 Output Shape              Param #
=================================================================
dense_52 (Dense)             (None, 1068)              1141692
_________________________________________________________________
dense_53 (Dense)             (None, 2136)              2283384
_________________________________________________________________
dense_54 (Dense)             (None, 1068)              2282316
=================================================================
Total params: 5,707,392
Trainable params: 5,707,392
Non-trainable params: 0
```

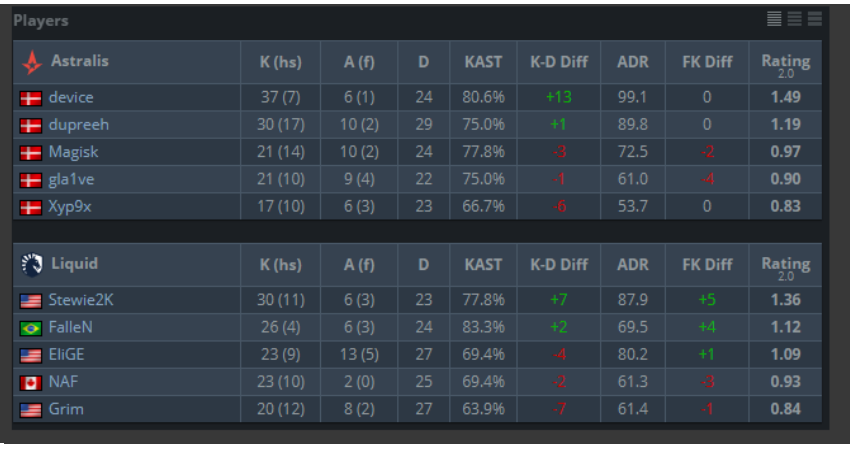
Figure 8: summary of medium model with L2 regularization

5. Using the Model

I created a couple functions so that I could simply input two lists of players, just using their names as strings, and then print the prediction my model serves for the game. Let's see what this model thought of those games from section 3.1:



```
Team 1:
EliGE      1.03
NAF        1.03
Stewie2K   0.87
FalleN     0.89
Grim       0.93
Agregate Rating: 4.75
Team 2:
device     1.21
dupreeh    1.14
Xyp9x      1.08
gla1ve     1.04
Magisk     1.13
Agregate Rating: 5.6
```



Players

Astralis	K (hs)	A (f)	D	KAST	K-D Diff	ADR	FK Diff	Rating 2.0
device	37 (7)	6 (1)	24	80.6%	+13	99.1	0	1.49
dupreeh	30 (17)	10 (2)	29	75.0%	+1	89.8	0	1.19
Magisk	21 (14)	10 (2)	24	77.8%	-3	72.5	-2	0.97
gla1ve	21 (10)	9 (4)	22	75.0%	-1	61.0	-4	0.90
Xyp9x	17 (10)	6 (3)	23	66.7%	-6	53.7	0	0.83

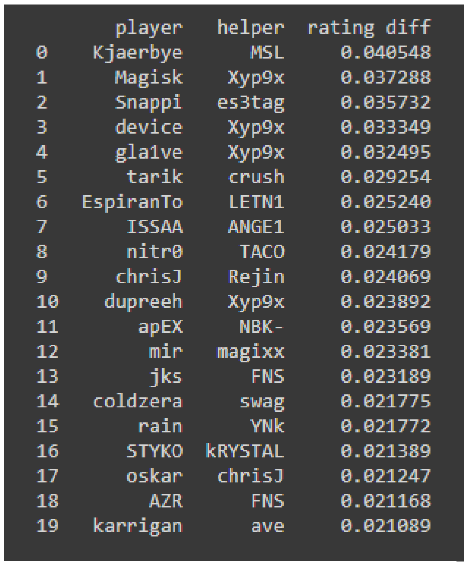
Liquid	K (hs)	A (f)	D	KAST	K-D Diff	ADR	FK Diff	Rating 2.0
Stewie2K	30 (11)	6 (3)	23	77.8%	+7	87.9	+5	1.36
FalleN	26 (4)	6 (3)	24	83.3%	+2	69.5	+4	1.12
EliGE	23 (9)	13 (5)	27	69.4%	-4	80.2	+1	1.09
NAF	23 (10)	2 (0)	25	69.4%	-2	61.3	-3	0.93
Grim	20 (12)	8 (2)	27	63.9%	-7	61.4	-1	0.84

Figure 9: model prediction vs reality for game

Compared to the previous model, this model keeps rating to more average values, it resists giving players higher ratings like 1.5, even though a rating around 1.5 isn't uncommon for at least one player in a game. I think this is because while the previous model could safely assume ONE player would get a high rating, this model must also say WHICH player, which is much harder to do so it just gives a conservative estimate. Still, this model isn't too bad, and correctly predicts which team would have the higher agregate rating (Astralis).

5.1 Finding the best teammate pairs.

A convenient feature of the model is that it can serve predictions on incomplete teams. I used this feature to predict the ratings of pairs of players, leaving the rest of the team and the other team empty. I created a list of every player that played over 300 games in the dataset. For each of these players I listed all of their teammates they played over 150 games with and predicted the rating of them with each of these players. I chose this high threshold of games because some players act as stand-ins for only a short period of time, and if for that time the team plays easier games it will show that player as providing great chemistry to his team.



```
        player   helper  rating diff
0     Kjaerbye      MSL     0.040548
1       Magisk    Xyp9x     0.037288
2       Snappi   es3tag     0.035732
3       device    Xyp9x     0.033349
4       gla1ve    Xyp9x     0.032495
5        tarik    crush     0.029254
6    EspiranTo    LETN1     0.025240
7        ISSAA    ANGE1     0.025033
8       nitr0      TACO     0.024179
9       chrisJ    Rejin     0.024069
10     dupreeh    Xyp9x     0.023892
11        apEX     NBK-     0.023569
12         mir   magixx     0.023381
13         jks      FNS     0.023189
14    coldzera     swag     0.021775
15        rain      YNk     0.021772
16       STYKO  kRYSTAL     0.021389
17       oskar   chrisJ     0.021247
18         AZR      FNS     0.021168
19    karrigan      ave     0.021089
```

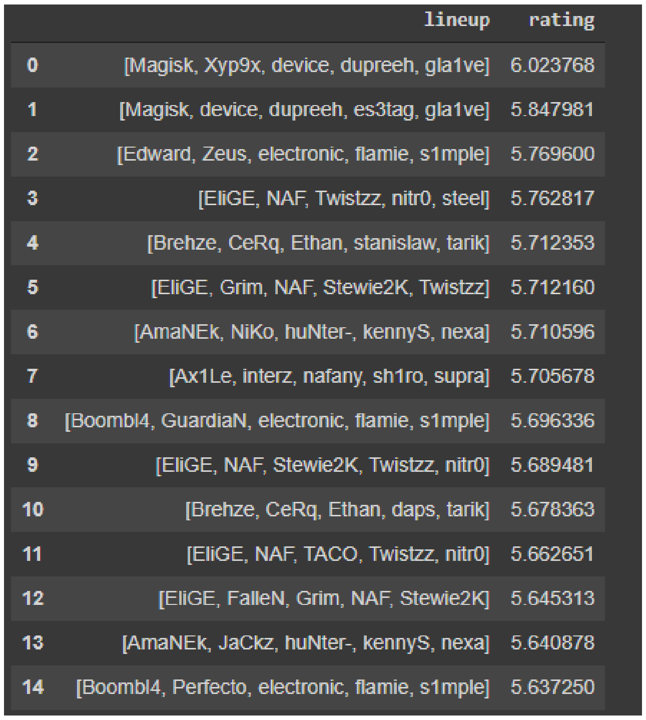
Figure 10: Best chemistry duos

Figure 9 contains the top 20 results of this process. The player column contains the player that had his rating improved, and the helper column is the teammate which improved the rating. The rating diff is the rating the player got in the pair, minus the average rating the player got with all his teammates. The top pair, MSL and Kjaerbye, played together for a stint in 2018 on a team called North. They won several tournaments, but the team quickly fell to middling results when MSL left. A player that appears in the helper column a lot is Xyp9x (pronounced Zipix). He is a player on Astralis and he took a hiatus from the game in 2018. Without him Astralis fell from absolutely dominating the game to "just pretty good". We can see the model has captured how much better all his teammates do when he is with them. Based on these results it seems like the model is capable of modeling some chemistry between players.

5.2 The best line-ups

Similar to what I did in section 5.1, I also got a list of every lineup that appeared in the dataset, that played more than 30 games together. I then predicted the rating for every lineup against an empty team and kept track of the aggregate rating. I put the lineup and aggregate rating into a pandas dataframe and sorted it.

Figure 10 shows the top 15 lineups from this dataframe. These teams do correspond to the most dominant teams from the time period of my dataset. The top two teams are both Astralis, with the second one containing es3tag, their substitute/map-specialist player. An interesting result is how many changes the Team Liquid (the lineups containing NAF and EliGE) has gone through while still being a top team. To me this shows that NAF and EliGE form the true "core" of team liquid and are probably the most important pieces to the teams success, which they had a lot of in 2019 when they dethroned Australis to briefly become the best team in the world.



	lineup	rating
0	[Magisk, Xyp9x, device, dupreeh, gla1ve]	6.023768
1	[Magisk, device, dupreeh, es3tag, gla1ve]	5.847981
2	[Edward, Zeus, electronic, flamie, s1mple]	5.769600
3	[EliGE, NAF, Twistzz, nitr0, steel]	5.762817
4	[Brehze, CeRq, Ethan, stanislaw, tarik]	5.712353
5	[EliGE, Grim, NAF, Stewie2K, Twistzz]	5.712160
6	[AmaNEk, NiKo, huNter-, kennyS, nexa]	5.710596
7	[Ax1Le, interz, nafany, sh1ro, supra]	5.705678
8	[Boombl4, GuardiaN, electronic, flamie, s1mple]	5.696336
9	[EliGE, NAF, Stewie2K, Twistzz, nitr0]	5.689481
10	[Brehze, CeRq, Ethan, daps, tarik]	5.678363
11	[EliGE, NAF, TACO, Twistzz, nitr0]	5.662651
12	[EliGE, FalleN, Grim, NAF, Stewie2K]	5.645313
13	[AmaNEk, JaCkz, huNter-, kennyS, nexa]	5.640878
14	[Boombl4, Perfecto, electronic, flamie, s1mple]	5.637250

Figure 11: The best lineups

6. Further Work

As I think I have shown in the previous section, the model is not useless, however, it may not be as useful as just doing normal statistics if you were interested in getting some measure of chemistry between players and teams. The advantage of deep learning is the ability to take in a lot of information and model it, so I think adding information to the model is one direction to take this in the future.

For example, more recent example is likely to be more predictive than examples from further in the past, so some way of passing in information about how long-ago example games were played should be investigated. It also seems like player chemistry should depend heavily on how long a set of players have played together. The longer they've been together, the more chemistry they can build, one would think. It would be very interesting to have some way to input how long a roster or pair of players have been together to the model. In conjunction with this I could constantly collect and train on new data from HLTV to keep the model accurate.

Since my simple, 10-output model did a better job at predict more extreme ratings, I should investigate if it is possible to combine that model with my more complicated one that could predict ratings for individual players. My intuitive idea is that the first model could predict the spread, and the second model could predict the individual ratings to establish the order of players to assign to the spread, or some mathematical combination of the spread prediction and the individual prediction.

During this project I could not think of a way to use the model to find information about how players play when certain players are present on the other team, in other words how rivalry affects players. Since I was initially interested in both rivalry and chemistry this should be investigated.

7. Conclusion

During this project I:

1. Created a JavaScript program to call an API and download a custom dataset from an online database of professional CSGO games.
2. Wrote a script to turn this dataset into a representation usable for machine learning.
3. Experimented with many different model architectures and two different data representations using TensorFlow.
4. Implemented a custom TensorFlow loss function, as well as other TensorFlow features like early stopping.
5. Used a model to discover interesting information about the chemistry of professional CSGO teams.

My final model ultimately seemed to do a decent job of modeling chemistry in professional CSGO based on my experiments with it.

I learned a ton about implementing machine learning from this project, from data collection to model evaluation. I am especially proud of that sigmoid-filtered loss function I implemented. And I had fun using my model to investigate chemistry in professional CSGO.

All of the code for this project can be found at: https://github.com/jamesNWT/player-chem-ml-stack, although it's a bit of a mess.

8. References

NBA Game Predictions Based On Player Chemistry, Prastuti Singh and Yang Wang, retrieved from: http://cs229.stanford.edu/proj2019aut/data/assignment_308832_raw/26645648.pdf