Exploring Winner Predictions and Probabilities in Real-Time Strategy Games via Neural Networks

Author: Andrew Jin
Advisor: James Glenn

May 2, 2019
# Contents

1 Abstract ................................................................. 2

2 Introduction and Background ........................................... 3

3 microRTS ................................................................. 4
   3.1 Game Description .................................................. 4
   3.2 Data Collection and Features ..................................... 5
   3.3 Neural Networks .................................................... 6
      3.3.1 Simple Neural Network ........................................ 6
      3.3.2 Unit Composition Neural Network ......................... 7
      3.3.3 Probability-based Neural Networks ......................... 8
      3.3.4 Model Calibration Evaluation .............................. 12
   3.4 Conclusion and Future Work ..................................... 13

4 Starcraft II ............................................................. 14
   4.1 Game Description .................................................. 14
   4.2 Data Collection and Features ..................................... 15
   4.3 Neural Networks .................................................... 16
      4.3.1 Simple Neural Network (NN 1.0) .......................... 16
      4.3.2 Adding Game Loop, Match Making Rating, and Race to the Simple Neural Network (NN 1.1) .................... 17
      4.3.3 Model Calibration Evaluation .............................. 18
      4.3.4 NN 1.1 Performance on MMR-Focused Data ............... 18
   4.4 Conclusion and Future Work ..................................... 20

5 References ............................................................. 22
1 Abstract

This paper explores in depth on the utilization of neural networks to predict winners and win probabilities during live matches of real-time strategy games, microRTS and StarCraft II. Although there has been considerable accomplishment for artificial intelligence in turn-based games such as chess and Go, real-time strategy games have been a major obstacle for researchers. An AI agent needs to process a vast observation space and an often even larger action space without the cushion of thinking during its turn. Even predicting just the outcome of a game can be very difficult. Those who consider themselves professionals of StarCraft II can still often be wrong in predicting a winner of a match despite them having full observability of the game. Despite this, little research has been conducted to calculate win probabilities of players in live real-time strategy games.

We begin our project by building and evaluating classification and probabilistic neural networks on a simplified implementation of StarCraft designed for AI research, microRTS. Using simple scalar environmental inputs and a sequential neural network with one hidden layer, we built a well-calibrated and accurate classification model with a halftime prediction accuracy of 90 percent. Then, we extended the architecture of our neural network to predicting win probability for StarCraft II. Using a multitude of scalar SC2 environment inputs and a 2-hidden layer sequential neural network, we built a well-calibrated classification model with a fairly accurate 70 percent halftime prediction, considering most games do not play until completion. We also analyze the effect of baseline skill-level and skill disparity on the accuracy of our model. Finally, we conclude that with more abundant data and processed inputs of unit composition and positioning, we could likely improve upon the predictive power of our models.
2 Introduction and Background

Real-time strategy games, such as Blizzard’s StarCraft II, are fast paced war simulation games where players have to manage their economy, control many units at a time, and deal with uncertainty about the opponent they are facing, whether it is the location of the opponent’s units or the strategy that he/she is going with. The keyboard and the mouse is the portal between players’ minds and their respective units. Since the game is played in real time, speed and effective multitasking abilities are imperative. This is especially true late in the game of StarCraft II, when players must control hundreds of units and dozens of buildings. Professional StarCraft II players average up to 300 to 400 actions per minute. Although it is complicated and requires large amount of multitasking, the goal of most RTS games are simple: efficiently allocate resources into building an army in order to defeat the opponent’s base.

For many years, turn-based strategy games such as Checkers, Chess, and Go have been used as major platforms to assess the performance and skill of artificial intelligence programs. Using machine learning techniques like supervised and reinforced learning, artificial intelligence programs have mastered these turn-based games and can beat pro human players pretty handily. However, real-time strategy based games have widely been considered by computer scientists as one of the most difficult challenges in the computational intelligence of games, as programs no longer have the time to think and compute actions in turns, but need to constantly process a vast observation space and compute actions in an even larger action space in real time. The large action space is a byproduct of the many different types of units and buildings, which each have their own unique set of actions. Since games typically last for thousands of frames, the consequences of a player’s actions may not be realized until much later in the game. The sheer magnitude of both the observation space and action space of StarCraft II has been a major obstacle for those interested in StarCraft II AI.

As such, developing bots for RTS games such as StarCraft involves a very large amount of time, engineering, and extensive processing power, which often relegates research aspects to a second plane. Thus, microRTS, a much simpler Java implementation of StarCraft, was created to motivate research in the basic research questions underlying the development of AI for RTS games, while minimizing the amount of engineering required to participate. Since its release, numerous papers and projects have been published that study different approaches to developing strong AI agents and evaluating the game state. For example, game tree search in RTS games like microRTS and StarCraft with very large branching factors is a notoriously hard problem that Santiago attempts to address in 2013 using Monte Carlo Tree Search paired with a new sampling strategy he calls "Naive Sampling." In 2016, researchers explored the possibilities of using a convolutional neural network to evaluate the spatial game state of microRTS games.

The eSports industry is growing at an explosive rate, and with competitive professional matches comes pre- and live game betting. Predicting the outcome of a game is a challenging task. Even professional StarCraft II commentators often fail to predict the winner despite having a full access to the game state (i.e., not being limited by partial observability). Thus, this project aims to help fix this by training a neural network via supervised learning to predict the winner/win probability of a given player during a competitive 1v1 real-time strategy match. We begin with a simplified approach of this problem by developing such a neural network for microRTS, then use a similar structure to predict for StarCraft II.

The ability to predict the outcomes of matches between players has become increasingly important, and this project may shed light on how to quantify likelihoods of victory for e-sports participants. Additionally, by training an AI agent to predict win probability rather than to competitively play the RTS game, we eliminate some of the problems that negatively affect existing game-playing bots. In terms of the observation space, the learning agent can observe all features of the game and is not hindered by
the “fog of war” game mechanic that is typical of RTS games.

3 \textit{microRTS}

3.1 Game Description

\textit{microRTS} is a small implementation of an RTS game, designed to perform AI research. The advantage of using microRTS with respect to using a full-fledged game like Starcraft is that microRTS is much simpler, and can be used to quickly test theoretical ideas, before moving on to full-fledged RTS games. Although the game comes with a built-in GUI that can be played by human players, \textit{microRTS} is really only designed to be played by bots, and the data from the replays we acquire are solely played by bots.\footnote{\textbf{Figure 1: Screenshot of microRTS, with explanations of the different in-game symbols.}}

In the \textit{microRTS} AI competition, bots play in two-player matches. Depending on the map, each bot starts with the same small set of units, and will need to handle all the aspects of the game (resource gathering, unit and building production, etc.). Based on the size of the map, games will run up to a fixed number of cycles (3000 for 8x8 maps, 4000 for 16x16 maps, 5000 for 24x24 maps, 6000 for 32x32 maps, and 8000 for 64x64 maps), after which the game will be considered a tie if no player has won. There are 10 game cycles per second, as each bot is given a computation budget of 100 milliseconds per game cycle.\footnote{\textbf{Figure 1} displays all the different types of units in the game, as well as the actions that the bot can take. The game map is a grid composed of 1x1 tiles that each can contain only one unit. Unlike Starcraft II, there is only one type of resource in microRTS (minerals) and players can spend minerals to build units. Bases train workers, which harvest minerals and build structures (bases and barracks). Barracks are used to train three types of army units: light, heavy, and ranged. Each type of army unit varies in cost, HP, damage, attack range, speed, and time to produce. In order for a bot to win, it must...}
destroy every unit that the opponent has, so that only its own units are on the map. Like Starcraft II, it is vital in this game to allocate resources strategically; building more workers during the early stages of the game can allow a player to boost their resource collection rate, which may help him/her in the future. However, by doing so they forgo the opportunity to build army units and can be vulnerable to attacks from the enemy.

3.2 Data Collection and Features

The purpose of our neural network is to approximate the value function $v^*(s)$, which represents the win-loss outcome of the game starting in state $s$ assuming perfect play on both sides. In practice, we have to approximate this function this with $v_{\theta}$, using a neural network with weights $\theta$. These states will be trained on state-outcome pairs $(s, w)$ using various neural networks. Our neural network will then be able to output either the winner or the individual probabilities of each player winning. The networks will not take into account previous observations, i.e., they predict the outcome from a single frame.

The dataset used for training our microRTS neural network is taken from the replays of the most recent 2018 microRTS AI competition. The competition is run as a round-robin tournament, where each bot will play against each other in a collection of different maps, twelve in total. They run five iterations of this round-robin tournament in order to get statistically significant results. There were twelve bots that participated in the tournament. In total, there were 7,700 games from the 2018 competition. However, there were some games that ended in draws because one player could not destroy all of the other player’s units within the time limit given. After discarding games that ended in draws, we ended up with 6,612 games that ended with a clear winner.

Each replay file is stored in an .xml file, which contains a large list of states of the game spaced out by approximately 5-10 game cycles. Game state snapshots are recorded more frequently as the game progresses because more is happening than the beginning, so more states are available during latter parts of the game. Each state in the list contains the time, the resources that each player currently holds, and a list of units that are currently on the map. Each unit in the list contains the type of unit it is, the player that owns the unit, the amount of HP it has, and its location on the map.

Predicting game outcomes from data consisting of complete games leads to overfitting because while successive states are strongly correlated, the regression target is shared the entire game. Longer duration games also contain more game states, which would bias our network towards longer games. To mitigate this problem, the authors of AlphaGo add only a single training example $(s, w)$ to the dataset from each game. Because we have significantly less data (6 thousand vs. 30 million games), we chose to sample 3 random positions from each game. As a result, for game $i$ we randomly add $(s_{i1}, w_{i1}), (s_{i2}, w_{i2}), (s_{i3}, w_{i3})$ to the dataset, generating a little under 20,000 positions. The raw replay files are not very easily parsable, so we implemented code that reads each file, grabs information from three random states from the file, and appends them to a readable .txt file for our neural net (microrts_simple_data.py and microrts_unit_data.py). The dataset is then randomly split into training and test sets using an 80/20 split, giving us roughly 16000 positions for our training set and 4000 positions for our test set.

Because of our limited amount of data, we first decided that grabbing a limited amount of features would be best to train our neural network appropriately. Thus, the first ten features we used for our neural network were the following: resource count of player 0, resource count of player 1, base count of player 0, base count of player 1, barracks count of player 0, barracks count of player 1, worker count of player 0, worker count of player 1, army count of player 0, and army count of player 1. The army count of each player was calculated by summing up all the light, heavy, and ranged units that each player owned. Features from the dataset were normalized from 0 to 1 and the label of each position was either 0 or 1.
indicating the winner of the state in that particular game.

Later on, we decided that although we did not have enough data or processing power to incorporate each individual unit positioning and health, extending our initial features to include army composition may help improve the accuracy of our neural network without adding too many more inputs. Thus, the fourteen features we used next for the neural network were the following: resource count of player 0, resource count of player 1, base count of player 0, base count of player 1, barracks count of player 0, barracks count of player 1, worker count of player 0, worker count of player 1, light unit count of player 0, light unit count of player 1, heavy unit count of player 0, heavy unit count of player 1, ranged unit count of player 0, and ranged unit count of player 1. Each army unit count was added as a feature rather than summed together. Like before, all features were normalized from 0 to 1 and the label of each position was either 0 or 1, indicating the winner.

3.3 Neural Networks

3.3.1 Simple Neural Network

As stated before, the inputs for our simple neural network consists of ten scalar features, which are the resources, bases, barracks, workers, and army of each player. We decided to build a classification neural network that will classify a state with either player 0 or player 1 winning based on the inputs it has been given. From keras, we used the Sequential model API to implement our neural network.

After some testing, we found that the best architecture for our neural network was to have the input layer, followed by one hidden layer of 100 nodes using the ReLU activation function. A dropout ratio of 0.5 is then applied to prevent overfitting of data. This is followed by the output layer of two nodes (one for each category). Since we are solving a classification problem, the activation function for this layer is set to softmax. The model is then compiled using categorical crossentropy as our loss function and using the Adam optimizer. Finally, the neural network is trained on our data using a batch size of 15 and 10 epochs.\(^1\)

When initially testing our model, we first checked to see how accurate the model was overall when predicting the winner for every state in the test set. However, we quickly realized this was not the best way to test our model, because our test set could consist of many states that are from the beginning of games, which would be much harder for our model to predict than states that are from the end of games. Thus, for each game state we grabbed, we included the time ratio of the state, which indicates the percentage of the game to completion. We do not include this in the training of the model as it would never have this information when determining the winner in a live game, but it is used when testing the model by putting each test set position in a bucket based on its time ratio. Ten buckets are used, each containing positions from the test set that have time ratios in [0, 0.1], [0.1, 0.2], ..., [0.9, 1.0]. Figure 2 shows the performance accuracy of our simple neural network.

\(^1\)The microRTS classification neural network used for the simple and unit composition models can be found in source code nnv1.py
The results from figure 2 is consistent with our expectations. At the start of any given microRTS match, both players begin with the same initial units. Hence, in the beginning of the game players should be even in chance of winning. As a consequence, at this stage of the game we should expect the predictions of the model to be essentially guessing the winner, with an expected accuracy of around 50 percent. Our graph supports this as the model achieves an accuracy of around 53 percent when the time ratio is between 0 and 0.1.

As the game progresses near the end, the divergence between the players become more prominent as a player is soon about to win. Thus, the higher time-ratio game states should be much easier to predict for the neural network, which is supported by the graph, as the simple neural network reaches close to 100 percent prediction accuracy when time ratio is between 0.8 and 1.0. It is also notable that the model is able to predict the winner more than 80 percent of the time when the game is halfway to completion, implying that the winner is almost decided by the halfway point.

### 3.3.2 Unit Composition Neural Network

As stated before, the inputs for our unit composition neural network consists of fourteen scalar features, which are the resources, bases, barracks, workers, light units, heavy units, and ranged units of each player. We use the same classification neural network used for the simple neural network using the Sequential model API from keras to implement our model. Each army unit type has its own unique properties in cost, HP, damage, attack range, speed, and time to produce. Thus, certain army units could be advantageous against one type of army unit but disadvantageous against another. We expected that when accounting for army composition, the model would have better prediction accuracy in earlier portions of the game when accounting for which player has a better combination of army units. Figure 3 shows the performance accuracy of our unit composition neural network compared to the simple neural network.
As expected, the unit composition neural network seems to perform slightly better than the simple neural network. In particular, the unit composition model performs much better than the simple model near lower time ratios, and the difference between their accuracy incrementally decreases as time ratio increases until they are essentially equal in accuracy when time ratio is $\geq 0.8$.

We believe this is because the unit composition model is able to predict better in the early and middle portions of the game based on differing army compositions of each player. Then, as the game progressively get closer to finishing, the difference in the players units are large enough that army count is enough for the neural network to know who is more likely to win. By the end of the game, we presume that the accuracy in both models are similar because the winning player has already destroyed a major portion of the loser’s units and is about to finish the game, simple enough for both models to predict near 100 percent.

3.3.3 Probability-based Neural Networks

One of the major drawbacks to training the neural networks we built is that the data and neural network is classification-based. Each state in the data is labeled with either 0 or 1, indicating the winner of the state of that particular game. Although the model determines its own individual win probabilities for each player, it classifies each position with the winner. The problem with this is that every position in a game is labeled with the winner of the game, even if the true probability of a player winning in the game state is nowhere near 100 percent. For example, in the very first position of a game, both players should theoretically have equal chance of winning, but because we know that one of the players ended up winning the game, we label this first position with the winner, when the true win probability of the position should be 50 percent. Another example is upsets: there could be games where player 1 won, but player 0 was actually winning for most of the game before player 1 made the comeback. However, the label of all the game states would have player 1 as the winner. These types of positions would bias our network, because it would be trained to believe that there is a reason the winner won in this state of the game, when in reality it is not true.

However, the problem is that we can never know the true probability of winning for any particular state in a game. The trained classification model can output its own perceived win probabilities of each player when given a position as it simply classifies the position with the player with higher win probability, we do not know if this probability is truly correct. Thus, in this section we explore different ways to estimate this probability, and train regression neural nets on these probabilities to see how well they perform. Although we cannot truly test how accurate a model’s outputted probability is, we can get a
good picture of its performance using statistical methods like calibration plots/reliability curves, which we explore at the end of this section.

### 3.3.3.1 Time-based Probability Model

The first probability-based neural network we explore is based on the time-ratio. There are a couple assumptions that we make when creating this model. The first is that in the beginning of a game, the probability that either player wins the game is 50 percent, and that at when the game is over, the probability that the winner of the game wins is 100 percent. Then, we assume that the probability of the winner of the game winning in any particular position ranges linearly from 50 percent to 100 percent, based on how close the game is to finishing. Since each position is originally labeled as 0 or 1 based on whether player 0 or 1 won, the time-based probability label we give the position is either between 0 and 0.5 if player 0 won or between 0.5 and 1 and player 1 won. Specifically, the function for the label of each position is as follows:

\[
    f(s, w) = \begin{cases} 
        0.5 + 0.5t_s & w = 1 \\
        0.5 - 0.5t_s & w = 0 
    \end{cases}
\]

where \( s \) is the state, \( w \) is the winner of the game, and \( t_s \) is the time-ratio of state \( s \).

The architecture that we use for this neural network is very similar to the architecture we used in nnv1.py. The input layer is followed by one hidden layer of 100 nodes using the ReLU activation function. A dropout ratio of 0.5 is then applied. This is followed by the output layer of one node. Since we are solving a regression problem this time, the activation function for this layer is set to sigmoid. The model is then compiled using mean squared error as our loss function and using the Adam optimizer. The neural network is also trained on our data using a batch size of 15 and 10 epochs.

One major drawback of using this model is that the win probability of a game is likely nonlinear. There are major events that happen in a game that influence a player’s probability to win; a player’s win probability does not simply steadily increase over time. Another drawback is that this model still does not fix the upset problem. In games with upsets, the model could be trained to think that the winner is constantly winning the entire game, when in reality the loser could have been ahead for a majority of the game until the end.

### 3.3.3.2 History-based Probability Model

The second probability-based neural network we explore is based on historical data. For each game state position in the data we have, we find all the positions in the data that match or are very similar to the position. Then, we find the mean of the winners of all the similar positions to give us the historical probability of player 1 winning from this state, the label for our position. If there are more positions with player 0 winning, the label will be closer to 0, and if there are more positions with player 1 winning, the label will be closer to 1. For example, for a beginning state where both player 0 and player 1 have 1 base and 5 workers, it should theoretically find all the states where both player 0 and 1 have 1 base and 5 workers, and calculate the historical probability that player 1 won from this position, which should be around 0.5. This would help our model account for upsets, as it would then be trained to know historically the probability of a a player winning from any given position.

The major drawback of this is that we simply do not have enough data to get good historical probabilities. The more inputs there are in a position, the harder it is to find other positions that have the exact same inputs. Thus, we have to make assumptions when grabbing similar states to compute historical probability. We do this by first finding the difference between player 0 and player 1’s individual inputs: calculate the differences in their resource, base, barracks, worker, light, heavy, and ranged counts.

\[\text{Time-based probability microRTS regression neural network source code can be found at nnv2.py}\]
Then, we arbitrarily classify these differences into different bins. Differences that are in the same bin will be considered the "same". For example, for difference in worker count, the bin label is calculated by the following function:

\[
f(d_w) = \begin{cases} 
-6 & d_w < -16 \\
-5 & -16 \leq d_w < -13 \\
-4 & -13 \leq d_w < -10 \\
-3 & -10 \leq d_w < -7 \\
-2 & -7 \leq d_w < -4 \\
-1 & -4 \leq d_w < -1 \\
0 & -1 \leq d_w < 2 \\
1 & 2 \leq d_w < 5 \\
2 & 5 \leq d_w < 8 \\
3 & 8 \leq d_w < 11 \\
4 & 11 \leq d_w < 14 \\
5 & 14 \leq d_w < 17 \\
6 & 17 \leq d_w
\end{cases}
\]

where \(d_w\) is the difference in worker count between player 0 and player 1. Similar functions are used for the other input differences.\(^{iii}\)

Once all differences are bin labeled, we find all the states that have the exact same bin labeled differences, and calculate the ratio of player 1 winning from these states. In total, there were 3634 out of 19836 positions with different bin labeled differences. After finding the historical probabilities, we simply add the column of probabilities back to the original data to be trained for the neural network. The problem with this method of data processing is that the bin labels for differences are very arbitrary and based on the user’s discretion. There is no real reason why a worker count difference of 4 is different from a worker count difference of 5 but the same as a worker count difference of 3. However, without this assumption we would not be able to use this method properly without enough data.

The network architecture is essentially the same as the Time-based Probability neural network, as we are still solving a regression problem. Source code for this model can be found at `nnv3.py`, where we run all three (historical-based, time-based, unit composition classification) models.

We evaluate both the time-based and historical-based models using the same method for the simple and unit composition models. For each game state in the test set, the model will compute a probability of player 1 winning. We then round the output to either 0 or 1, and use that as the categorical prediction for that particular game state. Figure 4 displays the performance accuracy of our time-based and historical-based probability models, along with our unit composition model performance for comparison. For accurate comparison, we run all three models on the same training set, and test them on the same test set.

\(^{iii}\)Historical-based probability data was processed using jupyter notebook and source code can be found at `history_prob_data.ipynb`
Figure 4: **Prediction Accuracy of History-based and Time-based probability microRTS Neural Network.** History-based probability regression model prediction accuracy (yellow) vs. time-based probability regression model prediction accuracy (light blue) vs. unit composition classification model prediction accuracy (dark blue) plotted against the stage of the game.

According to Figure 4, it seems that the unit composition classification model was the best at predicting the winner of each game state. The time-based model lagged behind it for time ratio \( \leq 0.4 \), but had essentially the same accuracy after. The history-based model did the worst; although it kept up with the unit composition model for time ratio \( \leq 0.3 \), it diverges and performs worse than both models after. This was surprising, as we expected the history-based model to do the best, but we believe this is because of our limited data, as we had to make assumptions in the data by grouping "similar" game states together, which can be arbitrary. This may also be due to not making the scope of our bin labels wider. For example, the worker difference bin labels could have extended to more then just -6 to 6, as there were worker differences in positions that were much larger than 17 and much smaller than -16. This may have led to the model being less sure if a position was more favorable for a certain player.

However, winner prediction accuracy is not the only way to measure the performance of our probability models. The goal of these models was to output a probability for a position that was as close to true probability as possible. Although we do not have the true probability for each position, we do have the true historical and time-based probabilities for each position in the test set. Thus, in Figure 5 and 6, we find the mean squared error for model’s outputted probability prediction based on true historical and time-based probability respectively.

Figure 5: **MSE of Neural Networks on Historical Probability.** MSE of History-based regression model probability prediction (yellow) vs. Time-based regression model probability prediction (light blue) vs. Unit Composition Classification model probability prediction (dark blue) on true historical probability plotted against the stage of the game.
Figure 6: **MSE of Neural Networks on Time-based Probability.** MSE of History-based regression model probability prediction (yellow) vs. Time-based regression model probability prediction (light blue) vs. Unit Composition Classification model probability prediction (dark blue) on true time-based probability plotted against the stage of the game.

It is trivial to see that the history-based regression model had the lowest overall MSE for true historical probability, and the time-based regression model has the lowest overall MSE for true time-based probability. However, it is interesting to note that in both figures, the MSE is a lot lower in the beginning and end of the game, but a lot higher near the middle of the game. This makes sense as it seems to show us that the true probability of a game state is easier to predict near the beginning (players are equal) and end of the game (one player about to win), but much harder to predict in the middle of the game, when the probability is likely more complicated to calculate. It’s also interesting to note that when the game is two-thirds the way done, the MSE drops off very quickly. This seems to indicate that games start to become more decided when the game is around 67 percent to completion.

Overall, the unit composition classification model performs the worst in both figures, which is expected because it was not trained for these datasets. This could hint that our classification model does not provide great win probabilities although it is better at predicting the winner. However, we do not know this for sure, as we do not know if these estimates of probability are close to the true probability. Another interesting thing to note is that in Figure 6, the unit composition classification model actually has a better MSE than the history-based regression model when the time ratio goes past 0.6. As the time ratio gets higher, the true time-based probability of a position becomes closer to either 0 or 1. All of this seems to indicate that the classification model gives a much more lop-sided win probability to the winning player earlier on, whereas the history-based and time-based models are much more conservative with its win probabilities, and thinks the game is closer than what the classification model thinks. As said before, this may be fixed by widening the scope and tightening the margins of our bin labels, but we would need more data to be able to find accurate historical-based probabilities.

### 3.3.4 Model Calibration Evaluation

Although we would not be able to know the true probability of one single position, one of the best ways we can measure the performance of our probabilistic models is by measuring probability calibration. When performing classification you often want not only to predict the class label, but also obtain a probability of the respective label. This probability gives you some kind of confidence on the prediction. Well calibrated classifiers are probabilistic classifiers for which the output of the its predicted probability can be directly interpreted as a confidence level. For instance, a well calibrated (binary) classifier should classify the samples such that among the samples to which it gave a predicted probability value close to 0.8, approximately 80 percent actually belong to the positive class.\(^\text{10}\)

Calibration can be assessed by using a calibration plot (reliability curve). A calibration plot
shows the proportion of items in each class for bins of predicted probability. Deviations from the identity function indicate a poorly-calibrated classifier for which the predicted probabilities or scores can not be used as probabilities.

In our case, because the outputted probability of our models corresponds to the estimated probability of player 1 winning, we put each of the networks probability predictions into 10 bins of bandwidth 0.1 (like we did with the time ratio). For each bin, we then calculate the percentage of times that player 1 actually ended up winning the game.\textsuperscript{iv} Figure 7 displays the calibration performance for the three models.

![Figure 7: Calibration Plot of the Three Compared Models](image)

Figure 7: Calibration Plot of the Three Compared Models Ratio of player 1 winning of History-based regression model (dark blue) vs. Time-based regression model (yellow) vs. Unit Composition Classification model (light blue) vs. Ideal/Perfect model (green) plotted against probability prediction of the models.

Observing Figure 7, we see that our unit composition classification model is actually the most calibrated of the three models, as it is the closest to the perfect model line. On the other hand, the calibration curves of both the history-based and time-based model look much more sigmoid like, indicating that they are underconfident in comparison to our classification model. This fits with our belief that both of the regression models believe that the games are closer than they actually are, although the history-based model seems to perform slightly better than the time-based model. For example, player 1 seems to win close to 100 percent of the games that the time-based model predicted with probability 0.8 and close to 90 percent of games that is predicted with probability 0.7, implying that although by rounding its prediction we would classify the correct winner, the model needs to be more confident in order for it to be more calibrated. In retrospect, the calibration of our regression models make sense, as one of the major drawbacks of labeling our training data with our estimated probabilities is that we are taking the matter of predicting probability into our own hands, and it is likely that our estimated probabilities are not close to the true probability of a position. The classification neural network, on the other hand, was able to relatively calibrate itself solely on the winner labels alone. However, the symmetry of the calibration curves of our regression models give us hope that we can re-calibrate our models.

### 3.4 Conclusion and Future Work

In conclusion, we found that by using very simple scalar inputs from \textit{microRTS} game states (resources, bases, buildings, workers, army), we could build a classification neural network that can quite accurately predict the winner early on in the game, with prediction accuracy around 85 percent at the halfway point. We also found that by adding in army composition, we were able to marginally improve our prediction accuracy by roughly 5 percent up until the game is 80 percent to completion, with prediction accuracy around 90 percent at the halfway point. This shows that understanding unit composition is important

\textsuperscript{iv}nnv3.py also contains code that calculates calibration plots for all three models.
to take into account when trying to predict which player is winning/about to win. We were also able to show that game states closer to the beginning and end of the game are easier to predict win probabilities for than game states in the middle of the game, as game states closer to the beginning of the game likely have a win probability of 0.5 and game states closer to the end of the game likely have an obvious winner. Lastly, we found that our unit composition classification model was likely the best of the three models we used to predict win probability, as the calibration curve of the model was the closest to the identity function/perfect model calibration line, implying that it was the more accurate in predicting actual win probabilities than the two probability regression-based models we explored. Although the classification model was much more confident than the other models in predicting a winner in a given game state, the proportion of times that player 1 won matched generally matched the outputted probability of the model. We have decided to use the classification based model as our main neural network when predicting for StarCraft II.

For future work, we would want to acquire a lot more microRTS game data, as we only grabbed data from the 2018 AI competition. One way would be to acquire and create our own microRTS bots, and create our own replays to analyze. Given more data, we would want to then take into account game state inputs that require more processing like individual unit positioning and health, and see if it improves our neural network even more. Theoretically, we would build a convolutional neural network to take in the raw board representation as an input to help improve our accuracy. With more data we could also improve our history-based model by finding more accurate history-based probabilities for each game state position, as history-based probability will probably be the closest to true probability as we can get. Lastly, we can improve our time-based and history-based probability prediction models by improving its calibration, as the reliability curve for both models are very sigmoid-like and symmetric, leading us to believe that calibrating the classification of these models with Platt scaling, for example, would help these models become more properly calibrated.

4 StarCraft II

4.1 Game Description

StarCraft II is the sequel of StarCraft, a timeless RTS game that takes place in a science-fiction universe ruled by three balanced races: Protoss, Terran, and Zerg (abbreviated as P, T, Z in context of matchup). Although there are many different ways to play the game, we will limit our focus to strictly competitive 1v1 games for the purpose of this project. The goal of a 1v1 match in SC2 is very similar to microRTS: strategically allocate resources into economy, tech, and army and either destroy the opponent’s base or get them to surrender. One of the main differences between StarCraft II and microRTS is that StarCraft II is (for now) mainly played by human players, and a player can decide to surrender if they think that they have lost, whereas a bot playing microRTS would continue to play until they no longer have units to use.

Players choose their race (Terran, Protoss, or Zerg) before the start of the game. When the game starts, each player (regardless of race) will have one base and twelve workers. Bases train workers (and posses other abilities depending on race) and workers are used to harvest resources and build structures. There are two types of resources: minerals and vespene gas. With these resources, workers can be used to build a large variety of different structures, including bases, army production buildings, and technology buildings used for upgrades. The map will usually contain many different spots with resources to "expand" to, or build a base near. Certain army production buildings produce certain

Sections 4.2-4.4 were mainly designed and implemented by James Chung. Please see his paper for more specific details.
army units, and some army units cannot be built unless the players has built a certain tech building, hence why "teching up" could be more beneficial to a player instead of building more army units if they want stronger units. Each race also has its own unique set of units, buildings, and abilities, but generally all three races have units with similar properties and are fairly balanced. Because of the many different options that player can go about playing the game and the many different matchups, players have developed a numerous amount of different strategies, from early expanding, to rushing (quickly building low level army units and attacking to win early), to investing in technology.

Another thing that makes StarCraft II unique from other strategy games like Chess and Go is that it is an imperfect information game. A game mechanic known as "fog of war" limits the vision of each player to a small radius around each of their units and buildings. Thus, each player has limited information about their opponent’s strategy, and it is necessary to actively explore the map in order to determine the opponent’s state. The fog of war mechanic is one factor that makes StarCraft II such a complicated game for AI.

4.2 Data Collection and Features

To train our StarCraft II neural network, we collected 10,000 1v1 ladder matches on version 4.8.3 using Blizzard’s SC2 API, sc2client-proto. After throwing out games where players would surrender immediately upon entering the game (which would skew our neural network), we were left with 8,466 usable games. We preprocessed each replay using a modified implementation of pysc2-replay, which was a framework built to access data from replays on top of DeepMind’s recently published StarCraft II API, pysc2. However, there were many limitations that we encountered when trying to access data from replays. First, the game state needs to be constructed from past game states, so we needed to construct all prior game states before reaching to the game state that we desired. This was cumbersome because although we only wanted to grab 3 or 4 game states per game, we needed to view the entire replay to grab them. Additionally, the pysc2 API only allowed us to scrape data from one player’s perspective at a time, so we needed to view and construct each replay twice in order to construct the full game state from both players’ perspectives. For each replay, we scraped the game state every 1000 game loops, which corresponded to about every 45 seconds. We later only used 4 random game states per game to prevent overfitting.

Finally, after grabbing all the replay data and removing corrupt files that occurred from using multithreading to run multiple instances of StarCraft II at once, we were left with 4,304 valid replay files. At each time step, we recorded the following for both players: current minerals, current vespene gas, food used, food cap, food army, food workers, idle worker count, army count, warp gate count, larva count, Blizzard score, idle production time, idle worker time, total value of units, total value of structures, killed value of units, killed value of structures, collected minerals, collected vespene gas, minerals collection rate, gas collection rate, spent minerals, spent vespene gas, and amount of lost and used resources in army, economy, technology, and upgrades. In total, we had 86 inputs. Due to the limitations we faced, we were unable to include unit positioning or the specific units each player had. After grabbing four random states from each game, we had a total of 17,216 game states, with the labels of each game state being the winner of that particular game. Features were normalized to continuous values ranging from 0 to 1. We split our data 80/20 into training and test sets, giving us 13,773 game states for our training set and 3,443 game states for our test set.

vi The StarCraft II data scraping source code can be found at transform_replay.py, ObserverAgent.py, ScrapeSC2replays.py, combineReplayTxts.py, combineReplayTxts_all.py, merge_replays.py, and formsc2data.py
4.3 Neural Networks

4.3.1 Simple Neural Network (NN 1.0)

We first started out by simply using the 86 inputs we had first obtained from the replay data as described above. After some testing, we found that the best architecture for our neural network was to have the input layer, followed by one hidden layer of 800 nodes using the ReLU activation function. A dropout ratio of 0.5 is then applied to prevent overfitting of data. This is then followed by another hidden layer of 800 nodes with another dropout ratio of 0.5 applied. Finally, the output layer of two nodes (one for each category) follows. We believe that two hidden layers were necessary because of the many inputs and the likely complicated relationship between them. Since we are solving a classification problem, the activation function for this layer is set to softmax. The model is then compiled using categorical crossentropy as our loss function and using the Adam optimizer. Finally, the neural network is trained on our data using a batch size of 15 and 10 epochs.\footnote{The StarCraft II classification neural network used for NN 1.0 and NN 1.1 can be found in source code sc2nnv1.py}

Figure 8 displays the performance accuracy of our simple model plotted against the time ratio.

![Graph showing accuracy of Simple StarCraft II Neural Network.](image)

Figure 8: Accuracy of Simple StarCraft II Neural Network. Accuracy of predicting the winner is plotted against the stage of the game, expressed as a ratio of the game length. Results aggregated in .10 buckets.

Upon observing Figure 8, our first realization was that our simple StarCraft II neural network performs worse than our simple microRTS neural network. We believe this is because of the difference between how StarCraft II and microRTS games are played. microRTS games are played by AI agents and the bots are not considered defeated until every unit they own is destroyed; every game is played until completion even if the match is essentially decided earlier. On the other hand, StarCraft II games are played between human players who are matched up against each other using Blizzard’s ladder matchmaking system, and humans can decide to surrender if they believe that there is no chance to come back and beat their opponent. Thus, StarCraft II matches are likely to be relatively even until close to the end of the match, when a significant event occurs that causes a player to perceive that they have no chance in winning and surrender.

This hypothesis is supported by our graphs. In Figure 2 (Prediction accuracy of our microRTS neural network), the accuracy curve is much more logarithmic, implying that the game is decided much earlier on and easier to predict in later portions of the game. In Figure 8, the accuracy curve is much more
linear and performs worse in early stages of the game. When time ratio is $\leq 0.3$, the model predicts the winner correctly just a little over 50 percent of the time. However, as time ratio increases, the prediction accuracy improves in a more linear fashion, increasing approximately 4.3 percent every decile. In the last decile, the accuracy of the model dramatically increases by close to 10 percent, indicating that there must be some significant event or change that occurs in the game (and to the position input features) that allows the model to better predict the winner. One of the reasons why this may occur is that usually there is a large battle of armies near the end of the game, which ends with one of the players losing all of his/her units and surrendering.

### 4.3.2 Adding Game Loop, Match Making Rating, and Race to the Simple Neural Network (NN 1.1)

Along with the initial inputs we used for our neural network, we believed that by adding the time of the game, the Match Making Rating (MMR) of players, and the race of the players, we could slightly improve the prediction accuracy of our model. Strategies in *StarCraft II* are very dependent on the current stage or time of the game (which are referred as early, middle, and late stages). Some early game strategies are rushing or expanding very early, while a late game strategy may be to tech up to acquire the ability to spawn late game units. We also believed that by adding players’ MMR, the model would be able to predict better earlier on in the game, when skill difference between players are more indicative of winning. Lastly, although races are generally perceived to be balanced, in an even game between different races, their scalar inputs like supply or resource collection may actually differ because their mechanics differ. For example, in an even game between Protoss and Zerg, Zerg tends to have a larger supply than Protoss. This is because Zerg army units are generally weaker than Protoss units and need to rely on hit and run tactics along with good surrounds on the Protoss army to succeed. We believed that by adding the race category, the neural network would be able to learn the intricacies between the different matchups and perform better. Figure 9 displays the prediction accuracy of the modified model vs. the simple model.

![Figure 9: Prediction Accuracy of StarCraft II Simple Neural Network with Game Loop, MMR, and Race added. Prediction accuracies of NN 1.0 (blue) and NN 1.1 (yellow) plotted as a function against the stage of the game.](image)

Contrary to our initial expectations, adding game loop, MMR and Race did not significantly improve the prediction accuracy of our neural network. However, it slightly improved the accuracy for earlier stages in the game. Because the size of our dataset was relatively small, we believe that the neural network did not have enough data to learn the nuanced relationship between race, gameloop, and other
inputs. Additionally, Blizzard’s matchmaking system often pairs players with similar or equal MMR on the ladder, and so the difference between player MMR may not have been able to make a significant impact when training the neural network.

### 4.3.3 Model Calibration Evaluation

Just as we did for our microRTS models, we want to evaluate the accuracy of our StarCraft II neural networks’ probability predictions. Although we cannot know if our models’ probability predictions are truly correct, we can assess their reliability using calibration curves. Figure 10 displays the calibration curves of both NN 1.0 and NN 1.1, along with the identity function (considered the perfect model) for reference.

![Calibration Plot of the Simple Model (NN 1.0) compared to the Simple Model with Added Inputs (NN 1.1). Ratio of player 1 winning of NN 1.0 (dark blue) vs. NN 1.1 (yellow) vs. Perfect model (light blue) plotted against probability prediction of the models.](image)

According to the graph, the calibration curves of both networks barely deviate from the identity function, although NN 1.0 appears to deviate slightly more than NN 1.1. The MSE of the probability predictions of NN 1.0 is 0.0015, whereas the MSE of NN 1.1 is 0.00035.

This fits our belief that although both models demonstrate good calibration, adding the extra inputs did indeed slightly improve our simple classification model: NN 1.1 appears to be slightly better than NN 1.0 in both winner classification accuracy and win probability accuracy. We use NN 1.1 going forward in the next section.

### 4.3.4 NN 1.1 Performance on MMR-Focused Data

#### 4.3.4.1 Large MMR Difference Data

Because of the fact that Blizzard’s matchmaking system pairs up players with relatively similar MMR, we wanted to find games where the MMR difference between players were large to see if the model had a better prediction accuracy. Thus, we filtered only for games where the MMR difference between the players was greater than or equal to 300 and received 426 games in total. In StarCraft II, players are placed into 7 separate leagues based on their MMR ranging from Bronze to Grandmaster. A 300 MMR difference would be equivalent to a player from the top of a league playing against a player from the bottom of the same league. Figure 11 displays the prediction accuracy of NN 1.1 is used on the large MMR difference dataset compared to the prediction accuracy of NN 1.1 on the overall dataset.
Figure 11: **Prediction Accuracy of NN 1.1 on Large MMR Difference Games.** Prediction accuracy of NN 1.1 when used on games with an MMR difference $\geq 300$ (blue) vs. when used on all games (yellow).

Figure 11 shows that using our neural network for only games with large MMR difference significantly improved its accuracy, especially very early on. In the first decile of games, the model was able to predict the winner correctly 74 percent of the time on the MMR difference games compared to 48 percent of the time for general games. In the first decile of games, the in-game features of both players are usually the same, so NN 1.1 is likely choosing the player with the higher MMR to win the match. However, the accuracy sharply declines before going back up when time ratio is between 0.1 and 0.3. This suggests that the model is weighting other inputs as the game progresses which influence its decision. It may also suggest that the players with higher MMR are doing things in the game that are not being accounted for in our inputs, which makes sense since the features we are using are scalar macro-oriented values.

### 4.3.4.2 League-Separated Data

We then decided to explore how the baseline skill levels of players affect the prediction accuracy of our model. Based on the average MMR of the game, we separated our dataset into three categories: Bronze-Silver (BS) (MMR $\leq 2640$, 926 games total), Gold-Platinum (2640 $<$ MMR $\leq 3440$, 1769 games total), and Diamond-Master (DM) (MMR $>$ 3440, 1611 games total).

Usually, the winner of a match in lower league games tends to be the player that macros better. This could mean that the player manages economy better or is able to produce more units. On the other hand, players in a higher league game are usually both able to macro fairly well, and the outcome of the game comes down to their difference in strategies and how they perform in battles. Because our model takes in mostly macro and economy-oriented inputs, we believe that our model would be better able to predict the winner in lower league games than higher league games. Figures 12 and 13 display the comparison in prediction accuracies and calibration plots respectively between the three categories.
The first thing to note is that the calibrations of NN 1.1 when used on the three separate leagues are fairly close to optimal, as all three are generally pretty close to the identity function. However, when it came to prediction accuracy, BS games seemed to be the easiest to predict early on compared to the higher level leagues. BS games were slightly easier to predict than GP games, which were slightly easier to predict than DM games. Also considering that they were well calibrated, this seems to indicate that in general, Bronze and Silver games were easier to predict and more frequently reached positions where one player had a substantial economic advantage over the other. Diamond and Master games were harder to predict likely because players were more even in terms on of their macro and economy until later on in the game when significant events occur that cause a disparity.

4.4 Conclusion and Future Work

In conclusion, although our StarCraft II neural network did not predict as well as our microRTS neural network, we contribute this to the fact that microRTS games are played to completion whereas StarCraft II games are cut short when a player surrenders. Despite this, our neural network was still
able to increase its prediction accuracy linearly over time, and was able to predict correctly just over 90 percent of the time in the last decile of games before a player surrenders. It was also able to correctly predict 75 percent of the time when the time ratio is between 0.7 and 0.8. Our StarCraft II model was also well if not better calibrated than our microRTS model, indicating that its outputted probability predictions are precise. We were also able to show that by adding game loop, MMR, and race as inputs to our model improved its prediction accuracy and calibration slightly, although not as significantly as we expected. Lastly, we showed that the prediction accuracy of our model depends on the base skill-level and skill disparity of the players.

For future work, we believe that accounting for unit composition could help improve our neural network, as it did for microRTS. Many games in higher level StarCraft II games are won when a player is able to correctly scout what the other player’s strategy is and build units that counter the opponent’s. For example, an army of units that are strong against armored units can destroy an equal supply of armored units with ease. Under the assumption that we acquire more data, incorporating unit composition, positioning, and health may help our neural network achieve higher prediction accuracy earlier in the game because it would be able to predict the outcome of a future fight before it actually occurs. Training our model on more data could also help it better understand more nuanced trends in specific race matchups.
5 References


