A Reasoning-based Defogger for Opponent Army Composition Inference under Partial Observability

Hao Pan
QOMPLX, Inc.
1775 Tysons Blvd, Suite 800
Tysons, VA 22102
hao.pan@QOMPLX.com

Abstract

We deal with the problem of inferring the opponent army size and composition in Real-Time Strategy (RTS) games with partial observability. Using StarCraft®: Brood War®, we propose a methodology to dynamically estimate the number of production facilities the opponent has at any given time in a game, then in turn reason on how many army units the opponent will produce in the near future and of which type. We conduct case studies to demonstrate the effectiveness of the proposed methodology and compare the results with those from mainstream approaches.

Introduction

The first known use of the exact phrase "Fog of War" in text dates to 1896, described as "the state of ignorance in which commanders frequently find themselves as regards the real strength and position, not only of their foes, but also of their friends."(Hale and Society 1896). In a nutshell, the Fog of War (FoW) describes the uncertainty regarding one's own capability, the opponent's capability and their intent during a military operation. In RTS games, FoW is often simulated. In as early as 1989, (Setear 1989) identified the key element of simulating the Fog of War to be uncertainty. In this paper we focus on the game StarCraft because it not only provides the environment with partial observability, but also has large state and action spaces, making the game itself challenging for AI researchers to design algorithms to handle various types of uncertainty and make appropriate decisions(Ontanón et al. 2013; Vinyals et al. 2017).

There are attempts to tackle this challenge. (Hagelbäck and Johansson 2009) attempted to address pathfinding under Fog of War utilizing potential field. The authors argued that the bots equipped with Potential Fields can handle partial observatiblity caused by FoW well by gridifying the entire map and performing efficient exploration. In this way, a bot can manage to navigate in a partially unknown environment while at the same time search for enemies and choose which one(s) to fight and where to do so. To deal with the strategy selection problem under uncertainty, (Gehring et al. 2018) utilized LSTM encoding and recorded game states based on current observation every 5 seconds of game time. They computed q-value to decide which of the 25 build orders (BOs) to switch to. The build order is defined as the concurrent action sequences that, constrained by unit dependencies and resource availability, create a certain number of units and structures in the shortest possible time span (Churchill and Buro 2011). These 25 BOs are a drop in the ocean, since the game StarCraft offers a seemingly endless choice of BOs which depend on factors such as the faction a player sides with and which of the 45 unit types to train at a specific time. (Fjell and Møllersen 2012) worked on extracting BOs from replay files and employed Gaussian distribution to handle the uncertainty in the BOs caused by the FoW.

Defogging is equally as challenging as decision making under FoW. It should be noted that the defogger this paper concerns is about inferring the army size and the composition of the opponent (future state prediction) under partial observability. This is quite different than the forward modeling (Justesen and Risi 2017) used. In their paper, the authors proposed a novel approach to counter an opponent's BOs by incorporating a Genetic Algorithm into an online planning algorithm so that the optimal BO can be found. A forward model was constructed to predict the outcome of taking some actions in a given game state. Such an outcome was not influenced by opponent information but only by one's own side such as the available mineral/gas amount, currently completed number of certain units/buildings, etc. (Synnaeve et al. 2018) employed convolutional LSTM encoder-decoder neural-network models to infer and predict an opponent's army composition under partial observability, and the Huber loss function to evaluate the accuracy of the prediction. The results indicated that the win rates improved (by 5% in some cases) with the proposed defogger. However, the authors only compared the overall observed unit count as opposed to the count of each unit type.

In this paper, we propose a reasoning-based defogger to address the following challenges: 1) handling uncertainty under the fog of war; 2) inferring the opponent's army composition (unit types and corresponding quantities) accurately

and efficiently. In the sections to follow, we present the framework to reason about the opponent's army composition with ECDFs to deal with the uncertainty, an entropy measure to intelligently choose the sampling rate of the information, and a combat simulator to aid projection into the future. We then demonstrate that the proposed approach performs better than a few mainstream approaches and is able to handle multiple unit types.

Methodology

The uncertainty brought about by the Fog of War makes defogging difficult. In the game StarCraft, if one were to predict how many of unit X there will be at a future time point, there are several questions to consider: 1) how many production facilities (ones responsible for training unit X) does the opponent have; 2) does the opponent have any production facility which is ready to train unit X; 3) whether the opponent decides to train unit X (instead of training other types of units); 4) how reliable is the collected intelligence of the opponent so far. And if we do believe that the opponent is about to train unit X, what we are then mainly interested in is: a) when the previous unit X was trained, and b) how long does it take between the time the unit is completed and the time we observe the newly-trained unit. Aspect b) can be tricky to quantify. Sometimes an opponent would like to stage their units for defensive purposes before they have a sizeable army, and sometimes such waiting time is negligible, especially when the opponent is doing a rush build, which means moving units and attacking the enemy as quickly as possible. We find the latter is often the case based on our observations of games happening on the StarCraft AI ladder called BASIL[1].

The first step in addressing these questions is to turn to the previous games vs the same opponent if such games are available. For each of these games, we record the time history of the count of each unit type that we observe in game as well as the actual number obtained after the game was concluded. Then we form an empirical distribution function (ECDF) based on such time series data at each time point, so that we can describe the distribution reasonably well regardless of the number of data points. If at the current time, the observation falls well within the distribution described by the ECDF, we will simply use the distribution at the next time step for prediction purposes, assuming continuity in time. Now doing so relies on several assumptions: 1) the opponent is following the same build order; 2) we also use the same BO; 3) map factors such as base-to-base distances play an insignificant role.

Regardless of the use of ECDF, the next step is the inference of the time when we'll see the arrival of the next unit. The arrival time of any unit can be described as:

$$t_{arrival} = t_{start} + t_{train} + t_{traverse} + t_{wait}$$

where $t_{arrival}$ means the time at which we observe the unit for the very first time, t_{start} means the time when the unit starts to get trained, t_{train} is the time it takes to complete the training, $t_{traverse}$ is how long it takes for the unit to traverse from its birth place to our sight, and finally t_{wait} is the time the unit is waiting. To ease the inference on t_{start}, we assume that, when the opponent chooses to train a unit of a certain type, there are enough resources, and the opponent is not supply blocked. t_{train} is a fixed value and measured by logical steps[2]. $t_{traverse}$ is affected by the movement speed of a unit and the map factors. While the movement speed is fixed just as t_{train}, there is much variation in the base-to-base distances (which are a good indicator for determining how far a unit needs to travel from its home to the enemy's base) on different maps. And finally, we assume t_{wait} is often negligible.

All of the things above are for a unit from a single production facility. In StarCraft, a player can have more than one production facility. Inference on the number of the opponent's production facilities can be tricky but can aid the inference of the army composition. We do this by reasoning on the arrival times of units. For example, with the current knowledge being the opponent having only 1 production facility, and if we observe the arrival of 2 new units after a time period which is only enough to train 1 unit (for the Terran and Protoss factions, only 1 unit can be trained at a production facility at a time), it is reasonable to infer that the opponent has at least 2 production facilities.

Another important factor for inferring opponent army composition is the sampling rate. How often do we collect intelligence about the opponent? In StarCraft and with the help of BWAPI, a bot can communicate with the game on each frame which is roughly 42 ms long at the fastest speed setting. But is it necessary to do something on each frame? In StarCraft AI competitions, there are frame time limitations, which means one needs to care how much computation to do per frame without slowing down the game. On the other hand, if we sample the information infrequently, some valuable information may eventually slip away. How can one decide the frequency to do so? Here we use the information entropy, or more specifically, Rényi entropy, as it is the generalization of many other entropy measures such as Shannon entropy and Hartley entropy. Entropy is a way to measure the information content of a system. Once we compute the entropy value, we then apply a change point detection algorithm (Killick, Fearnhead, and Eckley 2012) to know whether a drastic change in entropy value happens. A change point detection algorithm, in general, tries to segment the data according to some statistical criteria such as penalised likelihood and quasi-likelihood. When such a change point is found, we'll respond by increasing/decreasing the sampling rate among the 4 levels: I) once per second, II) once per 2 seconds, III) once per 4 seconds, and IV) once per 8 seconds.

Putting all things together, we arrive at our proposed approach for defogging: Algorithm 1 where T is the total duration of a game (in seconds) and t is a specific time step. A few things should be noted here. First, some production facilities are capable of producing multiple types of units, and only one unit can be trained at a time. This affects both the inference about the number of production facilities as well as

[1]BASIL ranking page: https://basil.bytekeeper.org/ranking.html

[2]Build times: https://liquipedia.net/starcraft/Game_Speed

determining whether a certain unit type is going to be trained next. For the sake of simplicity, we assume the worst case, that the opponent is going to choose a unit type that would benefit their army value the most, provided their technology level (which is based on our scouted information) supports that.

Algorithm 1 Our proposed approach for defogging

for t in $1 : T$ **do**
  * Infer how many production facilities the opponent has considering all unit types the opponent has
  * Update the number of production facilities using scouted information
  for unit type i in all unit types **do**
    if there is any enemy unit of type i being newly observered/destroyed currently at $t_{current}$ **then**
      * Increase/reduce the count of unit type i by the number of emerging/dying ones (check unit IDs to avoid double counting)
    end if

    if this is a time point to sample **then**
      * Evaluate Rényi entropy
      * Apply change point detection algorithm
      * Determine sampling rate
      * Compute next time to sample t_{next}

      if ECDF is available **and** ECDF is applicable **and** the current count falls within the distribution described by the ECDF **then**
        * Forecast the count by computing the mean value of the distribution at t_{next}
      else
        if there is any newly-observed enemy unit of unit type i **and** we believe the opponent will train it next **then**
          * Estimate the potential start time(s) of the next new unit(s): $t_{current} - t_{traverse} - t_{wait}$
          * Forecast the completion time t_{cmpl} by adding t_{train} to the result above
          * Forecast the potential losses of unit type i until t_{cmpl} should a skirmish happen
          * Update the count of unit type i at t_{cmpl}
        end if
      end if
    end if
  end for
end for

At each time step t, we also ask ourselves whether a skirmish would take place and if so, how many of the units the opponent would lose. The simulation of such a hypothetical encounter wouldn't be too hard as it would involve only a few types of units and we know the timing of our army pushing out. Here we used a combat simulator called FAP.[3]

Another thing to note is that, for each unit type, we check the validity of the associated ECDF carefully by looking at two things: 1) whether the standard deviation is at most half the value of the mean; 2) whether the observed value falls within one standard deviation from the mean. If both results of these two tests are positive, we say the ECDF is valid and will utilize the ECDF to predict the count of the associated unit type. The amount of previous games used to form the ECDF can be tricky. In a case study we will show that only a limited amount of such games are optimal for the purposes here.

Case Study

For our first case study, the purpose is to demonstrate the inner working of our approach. For the sake of simplicity, we pitch our own StarCraft AI bot called Halo[4] against an opponent which only does a single build order, and we run 10 games on a single map (Destination) to minimize the influence of map factors. The opponent name is WuliBot and it is a competent bot which achieved the second place in the student division in the SSCAIT[5] 2016/17 tournament. WuliBot does a zealot rush build exclusively so we only monitor the count of the main unit of the build: *Protoss_Zealot*.

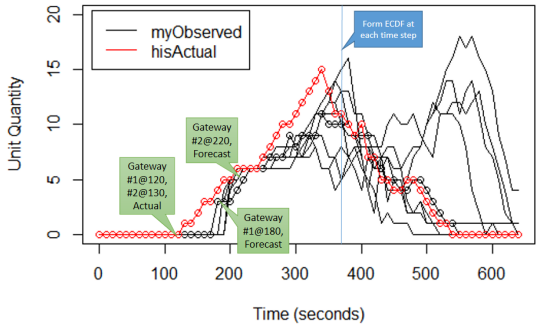

Figure 1: Count of the Protoss_Zealot unit from multiple games

Figure 1 shows the count of *Protoss_Zealot* from the 10 games we ran. The 10th game was used as the "current" one for forecasting purposes. Our observed and the actual counts of *Protoss_Zealot* in this game 10 are marked as black and red circles on Figure 1, respectively. With the metrics deciding whether the ECDF is applicable, we were able to ignore the ECDFs in the period between time points 50 and 65 where the variance is excessively large. On Figure 1 we also show a few of the inferred and the actual timings of *Protoss_Gateway* which is the production facility responsible for training *Protoss_Zealot*. At time point 180 we observed 3 *Protoss_Zealot*s, and after 40 seconds, we observed 2 more. It is impossible that there's only one *Protoss_Gateway* as a *Protoss_Zealot* takes 25 seconds to train and it would take more than 40 seconds for 2 new zealots to appear. Therefore we arrived at the conclusion that the opponent must have at least two *Protoss_Gateway*s at time

[3]FAP's github page: https://github.com/N00byEdge/FAP

[4]Halo's liquipedia page: https://liquipedia.net/starcraft/Halo

[5]SSCAIT stands for Student StarCraft AI Tournament and its homepage: https://sscaitournament.com/

point 220. This was in fact confirmed by our scouted information, as our scouting worker spotted 2 *Protoss_Gateway*s at time point 180. We then update the inferred timings of the two *Protoss_Gateway*s to time point 180. Efficient scouting (the scouting worker covering as much enemy territory as possible without dying) is vital to the inference here.

To examine the effect of the number of previous games on defogging, we started by having no previous games, and made one-step forward forecasts at each time step using the observed values from game 10. Then we treated game 1, games 1-2, and through to games 1-9 as the previous game(s) to build the ECDFs, and used game 10 to perform the forecasting. We used Root Mean Squared Error (RMSE) to measure how good the forecast is.

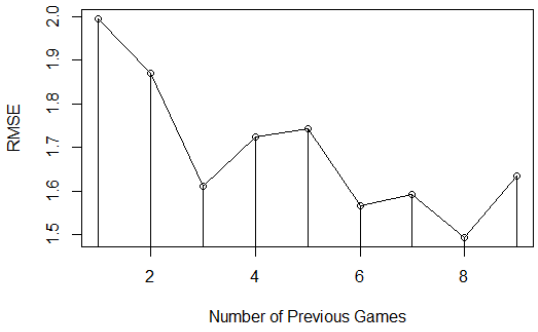

Figure 2: Evolution of RMSE as a function of the number of previous games

Figure 2 shows the evolution history of RMSE values as the number of the available previous games increases. The RMSE values decreases because the ECDF describing the distribution of unit count gets better as the evidence from the past accumulates. There is a slight upward trend in RMSE as the number of past games approaches 9. This is largely because the amount of noise/uncertainty is inevitably rising as the number of games increases. Taking advantage of a nice property of ECDF (that it works with low amounts of data), we apply a moving window here to limit how many of the previous games we consider.

Figure 3 shows the time history of entropy values for game 10. As we can see, there's a sudden increase around time point 180, indicating that the sampling rate should be raised. We were able to apply a change point detection algorithm to identify this very time point. And this is consistent with Figure 1 as there's also a sudden jump in *Protoss_Zealot* count around that time point. It is the correct decision to increase the sampling rate so that we have the count as accurate as possible.

The second case study aims at comparing our proposed approach with a few mainstream methods to perform forecasting for time series, namely, 1) ARIMA model; 2) neural network; 3) growth model. For the ARIMA model, we automatically choose the orders of the Auto Regressive (AR)

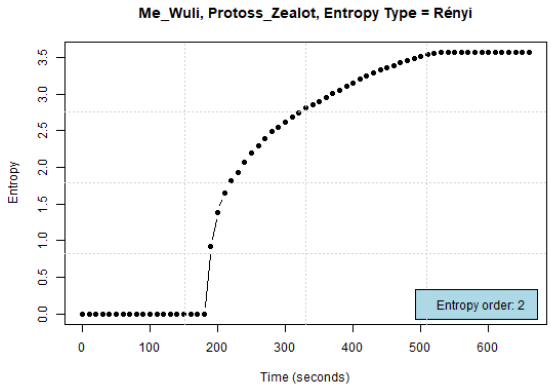

Figure 3: Time history of Rényi entropy

and Moving Average (MA) terms, and the number of differencing required to make the series stationary. For the neural network, we use a feed forward neural network called multilayer perceptron (MLP) as it is fairly conventional. We utilize an R package called "nnfor" which attempts to automatically specify autoregressive inputs and any necessary pre-processing of the time series. With the pre-specified arguments it trains multiple networks which are used to produce an ensemble forecast and a single hidden layer. For the growth model we pick the exponential type. There are two parameters to be estimated here: the initial state of the system and the growth constant. We avoid the other type of the growth model, the logistic type, since all the games run were nowhere close to either side hitting the supply max, which is not an ideal scenario for this type of model.

Before we performed the comparison, we also set up both a baseline model and a full-observability model in order to better examine how much the partial observability impacts the forecasts. The baseline model is one obtained with zero observability. Effectively the baseline model is trying to answer the question, "what army composition do we expect from an average Protoss opponent given a time point in a game?". We address this by running games using our own bot Halo with a fixed BO against all available Protoss opponents on the BASIL ladder. The baseline model is then constructed using the statistics from these games, mainly the mean and the variance. We do this for the two Protoss units which are often a staple: 1) *Protoss_Zealot*; 2) *Protoss_Dragoon*.

The full-observability model, on the other hand, was constructed using information as if there was full observability. We run games pitching our bot Halo against a single opponent named Locutus using sc-docker[6]. After each game, sc-docker produces a file named $unit\_events.csv$ which contains information such as location, unit type, and time associated with important events including $unitCreate$ and $unitDestroy$. One can then deduce the count of a unit at any given time from such information. Figures 4 and 5 demon-

[6]sc-docker's github page: https://github.com/basil-ladder/sc-docker. This is the version the BASIL ladder uses currently

strate both the baseline and the full-observability models for the two Protoss units. The curves denoted by games 1-3 represent the full-observability models for those games, respectively. Both figures demonstrated that the baseline model is working well. The upper bounds of the baseline model successfully encompassed the counts of the two Protoss units from the three full-observability models. In addition, the baseline model has a wide coverage in time. Games 1-3 ended much sooner compared to the average games depicted by the baseline model, as Locutus is notorious for its early/mid game aggression.

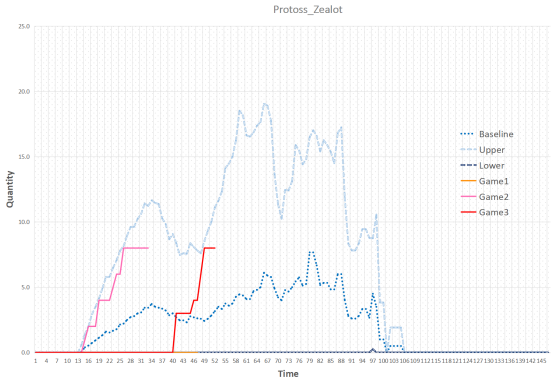

Figure 4: Baseline and full-observability models for the *Protoss_Zealot* unit

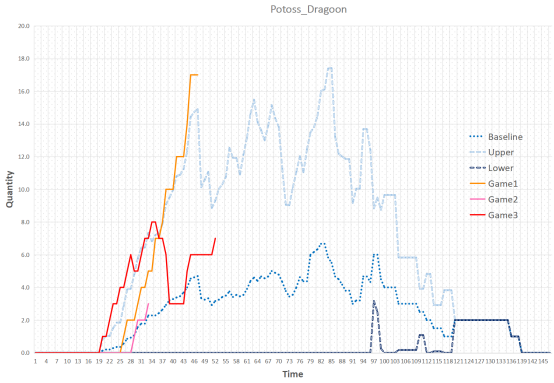

Figure 5: Baseline and full-observability models for the *Protoss_Dragoon* unit

Unlike the previous study, we now choose an adaptive opponent named Locutus which is performant and often achieves the top place on the BASIL ladder. We ran and chose a few games where Locutus executed a few of its main build orders. The build orders used were: 14 nexus in game 1; proxy 2-gate zealot in game 2; 3-gate goon in game 3.

Next we apply the three mainstream methods and our proposed approach to perform both one-step and twelve-step forecasts, as we care about both the short-term and long-term army compositions of the opponent. We then compared the forecasts produced from the four approaches, alongside those from the baseline model, with the "true values", or those from the full-observability model. Table 1 shows the summary of RMSE values from game 1 for the *Protoss_Dragoon* unit only, as Locutus didn't train any *Protoss_Zealot* at all. The results here suggest that our proposed approach was able to produce the most accurate forecast as the RMSE values are the smallest. Discussion on the causes of such results will follow next.

Table 1: RMSE using different approaches from game 1 for the *Protoss_Dragoon* unit

	ARIMA	MLP	Growth Model	Proposed	Baseline
			One-step forecast		
RMSE	3.5	1.8	1.9	1.7	4.7
			Twelve-step forecast		
RMSE	4.1	4.3	9.1	3.1	9.0

We discovered a few drawbacks associated with the alternative approaches. For the ARIMA model, forecasts are not ideal, as evidenced by Figure 6. The twelve-step forecast is indicated by the blue line, with a growing variance as time goes by. The ARIMA $(0, 2, 2)$ model fitted here is effectively applying a linear exponential smoothing which uses a moving average to create a forecast from a time series. In the case here, this is problematic, as the count of a unit can change drastically because it depends on so many factors such as combat effectiveness, the number of production facilities, the amount of available resources, etc. The moving average part simply cannot keep up. The one-step forecast is even worse, as indicated by the red dots, especially towards the end of the game. One large contributing factor to this is the fact that the quantity of the *Protoss_Dragoon* unit is zero from the start for a very long time (up until 28 time units). They "dragged down" the forecasts later on, because the moving average term of the ARIMA model $(0, 2, 2)$ is not sensitive to new trends. While for our proposed approach, such dependency exists, but the updated information from either our scout or real-time observations prevents us from being stranded by the past data.

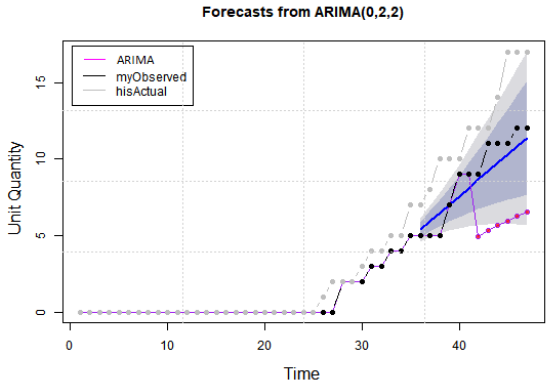

Figure 6: Forecasts from an ARIMA model

For the neural network model, there's a mandatory burn-

in period. In this case study, it required at least 10 data points for the neural network model to produce meaningful forecasts. Also, there's the issue of hyperparameter tuning where one needs to decide the number of layers and the number of nodes on each layer for a neural network model. However, the advantage of the neural network approach is that it would work for any game, whereas the proposed method might have to be redesigned for a different RTS game. For example, some games might not even have the concept of production facilities. Regardless of performing hyperparameter tuning, the computational cost is the highest using neural network models compared to other methods. While for our approach, it works right from the beginning without requiring hyperparameter tuning. Then as time goes by, it updates accordingly with minimal computational cost. Figure 7 shows the forecasts from the neural network model MLP. It is similar to the forecast from the ARIMA model earlier as it also predicts an upward trend. The variance associated with the forecast is much smaller, which is an improvement. The upward trend is a bit conservative, which results in a growing discrepancy between the forecast and the actual unit count.

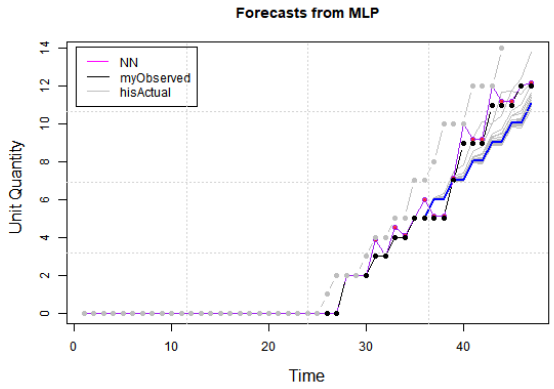

Figure 7: Forecasts from an MLP model

For the growth model, we had to stick with the exponential type as the logistic type is quite sensitive to the zero values in the early times, resulting in unsuccessful fitting of the model. While for our approach there is no need to choose/fit a model, as it is relatively model-free. Figure 8 shows the forecasts from the exponential growth model. The results are not meaningful, as the predicted values are unreasonably huge, well beyond the maximum number of *Protoss_Dragoon* a player can possibly have, not to mention that the fitting of the model to the training data set is poor. The model is overly simplistic and gives almost no consideration to limiting factors such as available supply and resources. One may argue that the other main type of growth model (the logistic type) may be applicable here despite its computational difficulty. However, logistic models assume that the growth rate decreases with population abundance, which is not necessarily true in StarCraft.

Figure 9 shows the result of our proposed approach. First we correctly captured the growth rate of the *Protoss_Dragoon* unit, as the slope of our forecast curve is

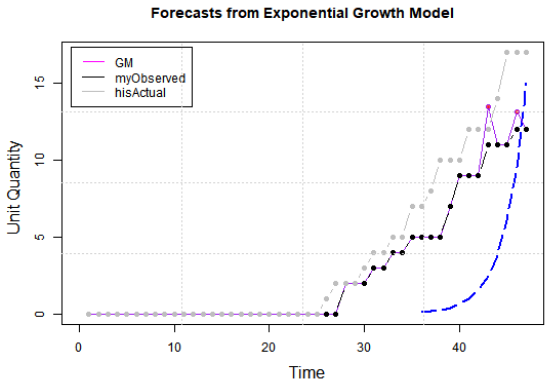

Figure 8: Forecasts from an exponential growth model

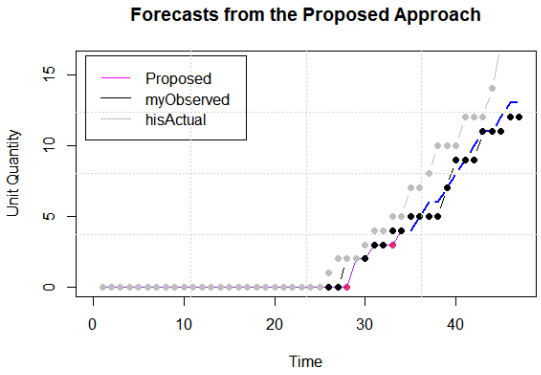

Figure 9: Forecasts from our proposed approach

roughly matching the one from the curve representing the true values. This is mainly because we correctly inferred the number of production facilities the opponent had, although there's a slight delay on the time the opponent started to train the unit. Also, we were successful in predicting the potential casualties incurred upon the opponent unit.

Table 2: RMSE using different approaches from game 2 for the *Protoss_Zealot* unit

	ARIMA	MLP	Growth Model	Proposed	Baseline
One-step forecast					
RMSE	2.1	0.8	0.9	1.0	2.9
Twelve-step forecast					
RMSE	1.3	1.4	132.0	0.8	4.7

Tables 2 and 3 show the RMSE values from games 2 and 3, respectively. In both of these games, Locutus went for a single type of unit exclusively for a really long time. The other type of unit was only added into the army mix in the last minutes of those games. Nonetheless, the presence of multiple unit types still poses a challenge to the defogging. With different build orders used by the opponent and a more

Table 3: RMSE using different approaches from game 3 for the *Protoss_Dragoon* unit

	ARIMA	MLP	Growth Model	Proposed	Baseline
One-step forecast					
RMSE	2.4	1.7	1.7	1.2	2.6
Twelve-step forecast					
RMSE	4.5	5.8	28639.0	1.1	2.1

complex army composition, our proposed approach was still able to produce a reasonable forecast with the minimum RMSE compared to values from other approaches in most cases. We were successful in maintaining the accuracy of the forecast by taking into account multiple unit types at the same time, thus preventing us from under-/over-predicting. For production facilities the Protoss faction possesses, it is not possible to train more than one unit at a time. This fact aided our inference on the number of production facilities the opponent had, and such an inference is an advantage other approaches do not have. Additionally in game 3, the casualty count of Locutus' *Protoss_Dragoon* unit was exceedingly high. But thanks to our consideration of potential combat outcomes, we were able to adjust our forecast accordingly.

Conclusion

In this paper we proposed a customized approach to handle the challenge of performing defogging under partial observability. We infer the counts of different types of units the opponent has by reasoning on several things, such as the number of production facilities, currently collected intelligence about the opponent, the outcome of a future skirmish, etc. We utilize ECDF to handle the uncertainty brought about by the Fog of War. We also employ information entropy to adjust the sampling rate properly. In our case studies we demonstrated the effectiveness of our proposed approach and illustrated its supriority over a few mainstream ones such as ARIMA, MLP, and growth models.

There are a few things that we'd like to work on in the future: 1) opponent build order classification/identification. Proper opponent build order classification would allow us to select which of the previous games should be used based on the BO identified in the current game, thus facilitating the forecast process; 2) opponent intent inference. Currently we assume that the opponent would choose the unit which would counter our army the best. In reality, things are more complex. For example, there can be multiple such units to choose from, or the opponent feints us by choosing to train sub-optimal units, so that we are led to situations that are hard to transition out of; 3) further reducing the delay in the forecast. Although we were able to achieve good RMSE values, there's still a gap in time between our forecast values and the actual ones. On top of the factors that we are already considering such as map terrain and unit speed, there are yet potentially important ones, into which we can dive deeper, such as unit waiting time.

Acknowledgements

The authors would like to specifically thank Nathan Roth (MSc) for his assisstance on the making of the authors' StarCraft bot Halo. The authors would also like to thank Dan Gant for his review on this paper and advice. The authors are equally thankful to Dennis Waldherr and his BASIL ladder, as it provided an invaluable testbed-like environment for this research.

References

Churchill, D., and Buro, M. 2011. Build order optimization in StarCraft. *The Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)* 14–19.

Fjell, M. S., and Møllersen, S. V. 2012. Opponent modeling and strategic reasoning in the Real-Time Strategy game StarCraft. Master thesis, Norwegian University of Science and Technology.

Gehring, J.; Ju, D.; Mella, V.; Gant, D.; Usunier, N.; and Synnaeve, G. 2018. High-level strategy selection under partial observability in StarCraft: Brood War. doi: https://arxiv.org/abs/1811.08568.

Hagelbäck, J., and Johansson, S. J. 2009. Dealing with Fog of War in a Real-Time Strategy game environment. 55 – 62. 2008 IEEE Symposium on Computational Intelligence and Games (CIG08). doi: 10.1109/CIG.2008.5035621.

Hale, L., and Society, A. M. 1896. *The Fog of War, by Colonel Lonsdale Hale ... Tuesday, 24th March, 1896*. Edward Stanford, 26 and 27, Cockspur Street, Charing Cross, S.W.

Justesen, N., and Risi, S. 2017. Continual Online Evolutionary Planning for in-game build order adaptation in StarCraft. 187 – 194. GECCO '17: Proceedings of the Genetic and Evolutionary Computation Conference. doi: https://doi.org/10.1145/3071178.3071210.

Killick, R.; Fearnhead, P.; and Eckley, I. A. 2012. Optimal detection of changepoints with a linear computational cost. *Journal of the American Statistical Association* 107'(500):1590–1598. doi: https://doi.org/10.1080/01621459.2012.737745.

Ontanón, S.; Synnaeve, G.; Uriarte, A.; Richoux, F.; Churchill, D.; and Preuss, M. 2013. A survey of Real-Time Strategy game AI research and competition in StarCraft. *IEEE Transactions on Computational Intelligence and AI in games* 5(4):293–311.

Setear, J. K. 1989. Simulating the fog of war. Research report, RAND Corporation.

Synnaeve, G.; Lin, Z.; Gehring, J.; Gant, D.; Mellla, V.; Khalidov, V.; Carion, N.; and Usunier, N. 2018. Forward modeling for partial observation strategy games - a StarCraft defogger. *Advances in Neural Information Processing Systems* 31:10759 – 10770. doi: arXiv:1812.00054.

Vinyals, O.; Ewalds, T.; Bartunov, S.; Georgiev, P.; Vezhnevets, A. S.; Yeo, M.; Makhzani, A.; Küttler, H.; Agapiou, J.; and Schrittwieser, J. 2017. StarCraft II: A new challenge for Reinforcement Learning. arXiv preprint, arXiv:1708.04782.