THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Supervised Learning of StarCraft 2 Game Prediction Using RNN

SEN LU
SPRING 2019

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Computer Science
with honors in Computer Science

Reviewed and approved* by the following:

Daniel Kifer
Professor of Computer Science and Engineering
Thesis Supervisor

Rebecca Passoneau
Professor of Computer Science and Engineering
Honors Adviser

*Signatures are on file in the Schreyer Honors College.

Abstract

Recent achievements by AlphaGo have inspired many interests in training sophisticated agents to excel in more challenging games. This paper aims to aid the reinforcement learning agent by providing it a more accurate predictor of game outcome which can effectively improve the reward function of the agent. The supervised predictor trained uses deep learning techniques, especially RNNs, on the replay data. While this is for StarCraft 2 replays specifically, the analysis and learning model of replay observations can extend to other video games with little modifications.

Table of Contents

List of Figures

Acknowledgements

Special thanks to my supervisor Dr. Daniel Kifer's kind and patient guidance.

Chapter 1

Introduction

1.1 Background

DeepMind's success in the board game GO [1] has proved reinforcement learning to be a useful and powerful mechanism in complex games with the astronomical number of states [2]. However, they were not fully successful when dealing with partially observed games. StarCraft 2 is a competitive Real Time Strategy (RTS) game that requires both macro- and micromanagement in a partially observed environment. This means that unlike GO, in which enemy's every single move is observed accurately and players take turns to move, StarCraft 2 requires the players to react to events and encounters in the game in real time despite the uncertainties caused by the "Fog of War." This has been a big challenge for a conventional agent. DeepMind has trained a memory-less supervised model to predict the game outcome and has achieved a 65% accuracy at 15 min mark of the game. However, it may be insufficient to use that model to support their reinforcement learning agent's value function as they noted. Furthermore, despite the most recent victory of DeepMind's AlphaStar over a professional StarCraft 2 player [3], AphaStar's access to the raw game states still backfires its credibility. The project of this paper attempts to train a better supervised-model for the prediction of the game only to better support the value functions of the reinforcement learning agent's training so that the game state can be pre-learned and thus reducing the reliance on the access to raw game states provided by the API.

1.2 Related Works

Many researchers have poured their interests into the RTS game StarCraft 2. Tstar by Tencent AI lab [4] has developed the first AI that learns a human-designed composition of an army in the game until AphaStar came, whereas some have also mined replays for supervised learning in build strategies [5] or used Bayesian models to do Opening prediction [6]. However, very little, if not none of any well-established game outcome predictor is found other than in DeepMind's work [2]. While it may seem less important considering the balance StarCraft 2 has, an accurate game outcome predictor will be extremely useful for the Multi-Agent Reinforcement Learning structure of AlphaStar.

1.3 Replays as Training Data

The data comes from the replays provided by the game company Blizzard. The replay of StarCraft 2 is a compact way of storing game history. Exploiting the fully deterministic nature of the game, a replay of StarCraft 2 minimizes its size by only recording the initial information and the events occurring throughout the game at a discrete time step. In other words, the replays provide the minimum required information to reproduced the battle rather than recording every details [7]. The training replays come from the game replays provided by the Starcraft 2 game platform's competitive matches. There were 800K anonymous replays available for download and the number of replays increases steadily. Pysc2 API, developed by the concerted effort of Deepmind and Blizzard Entertainment(the company which StarCraft 2 belong to), provides the un-rendered screen of the gameplay for training purpose as shown in the Figure 1.1

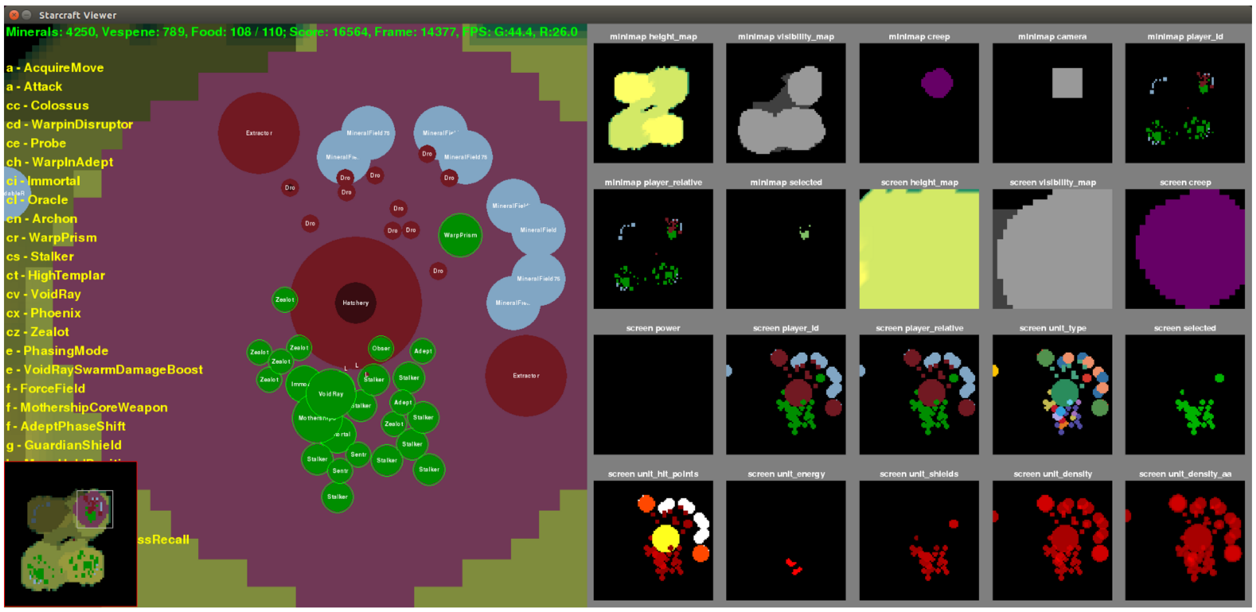

Figure 1.1: A screenshot of the screen generated by Pysc2 API. It provides each unit's position with a label and relative size in the circle.

While the replay provides both player's statistics (e.g., supplies, armies), only one player's view should be used to train the model as partial observability is a standard condition in a normal game. One observation at any given time consist of the followings:

- **Camera screen**: An unrendered camera view of the player pointing at a portion of the map. It consists of spatial and unit information. The resolution is 64 $\times$ 64.
- **Mini-map**: An overview of the entire battlefield. Regions with the vision of the player's unit will be highlighted while the rest will be obscured by the "Fog of War"(shaded). The resolution is also 64 $\times$ 64.
- **Player stats**: They are the numerical values of the resources and the supplies the player currently has. The values will be integers. Additionally, the actions available to the player is also part of the non-spatial stats.

1.4 Data Extraction from Replays

While the replay file 'x.SC2Replay' is very small, the extraction of replay observations are less friendly. To reproduce the exact same scenario, the extraction goes through the following stages.

1.4.1 Setting Up the Running Environment

The following must be set up before obtaining observations from replays:

- The main game engine version must match exactly with the replay file's game version.
- The map data for the replay are available.

- Related APIs are installed (e.g., PySC2).
- The Internet is available to connect to the game server.

1.4.2 Extracting Observations

Since the replay file is only a record of events, the replay can only start from the beginning and is not reversible. The local game engine sends every event to the game server and collects the resulting observations for rendering. This delayed communication and response become too significant overhead for data generator described in Figure 1.2.

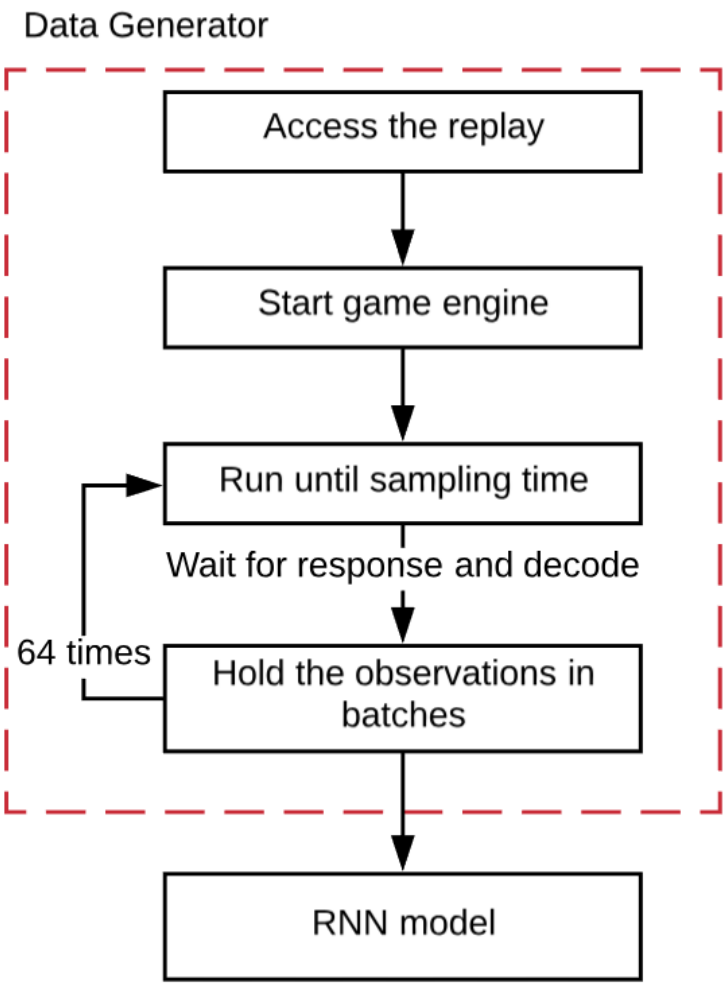

Figure 1.2: Version 1 generator with low space requirement but takes longer time.

Therefore, the replay data is extracted before the training to reduce the time cost as shown in Figure 1.3. However, the memory space that the data set used is nearly 200 GB after extraction (24.2MB/replay), and it was very cumbersome to move the data set around, although the training speed now depends on the speed of which the machine accesses the memory.

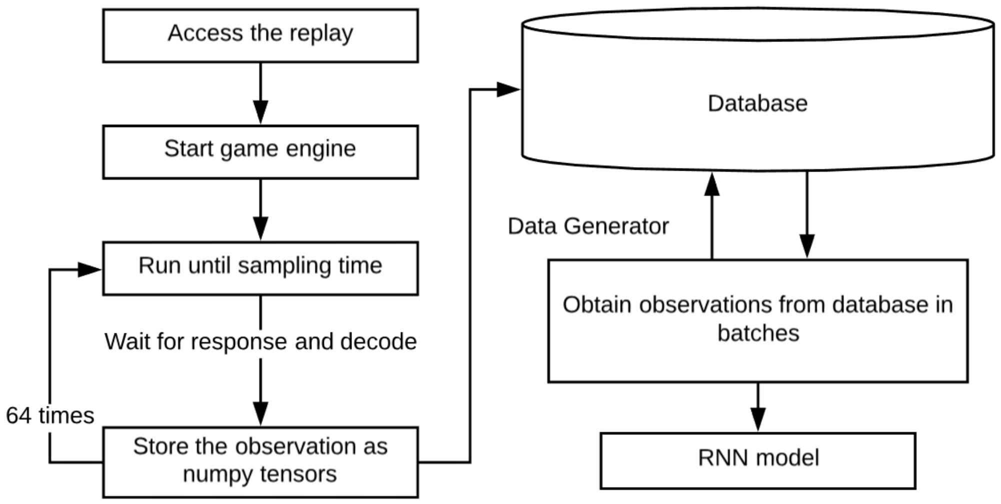
Access the replay
Start game engine
Run until sampling time
Wait for response and decode
64 times
Store the observation as numpy tensors
Database
Data Generator
Obtain observations from database in batches
RNN model

Figure 1.3: Version 2 generator with faster speed but takes a larger amount of space on disk.

Chapter 2

Existing Approach

2.1 DeepMind's FullyConv Architecture

The DeepMind's model was trained with mini-batches of 64 observations randomly taken from all replays (data) uniformly. Although the mini-map and screen capture the spatial features, specific stats on screen are fetched directly from the API provided by the game engine. The three components from one observation is first concatenated to create a state summary and then fed into the neural network shown in Figure 2.1 and 2.2.

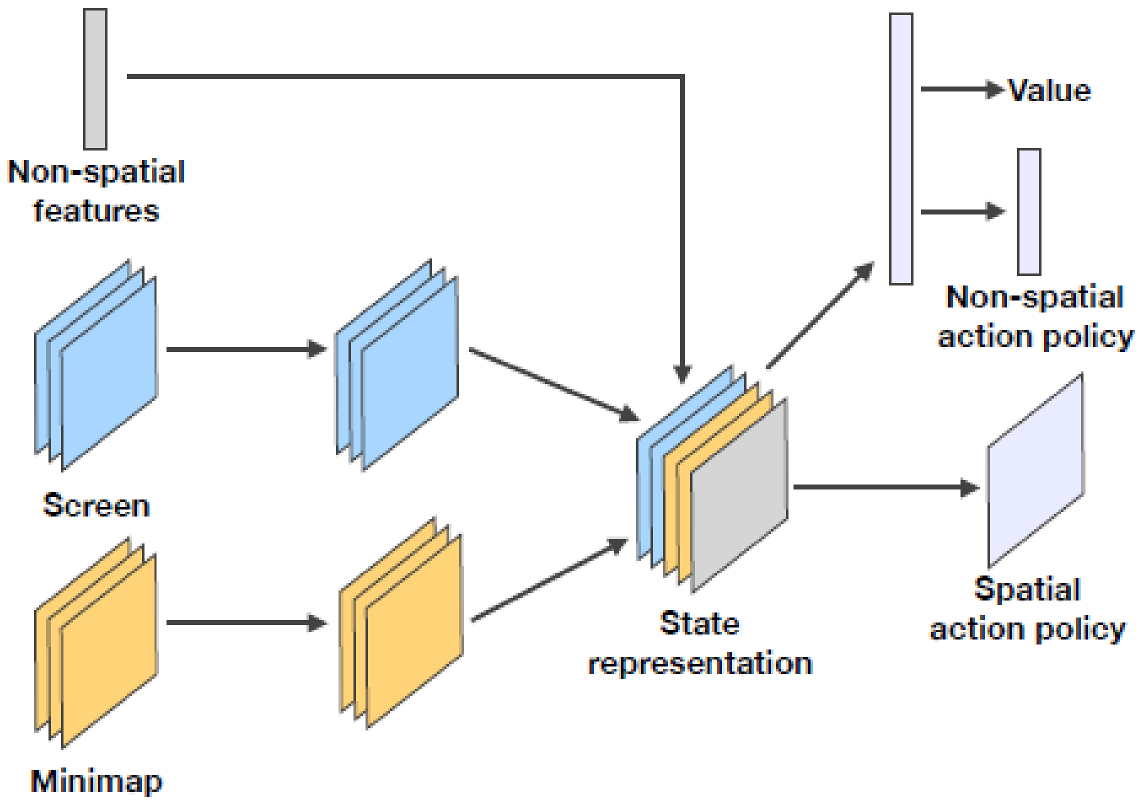

Figure 2.1: A visual representation of the neural network's flow[1]

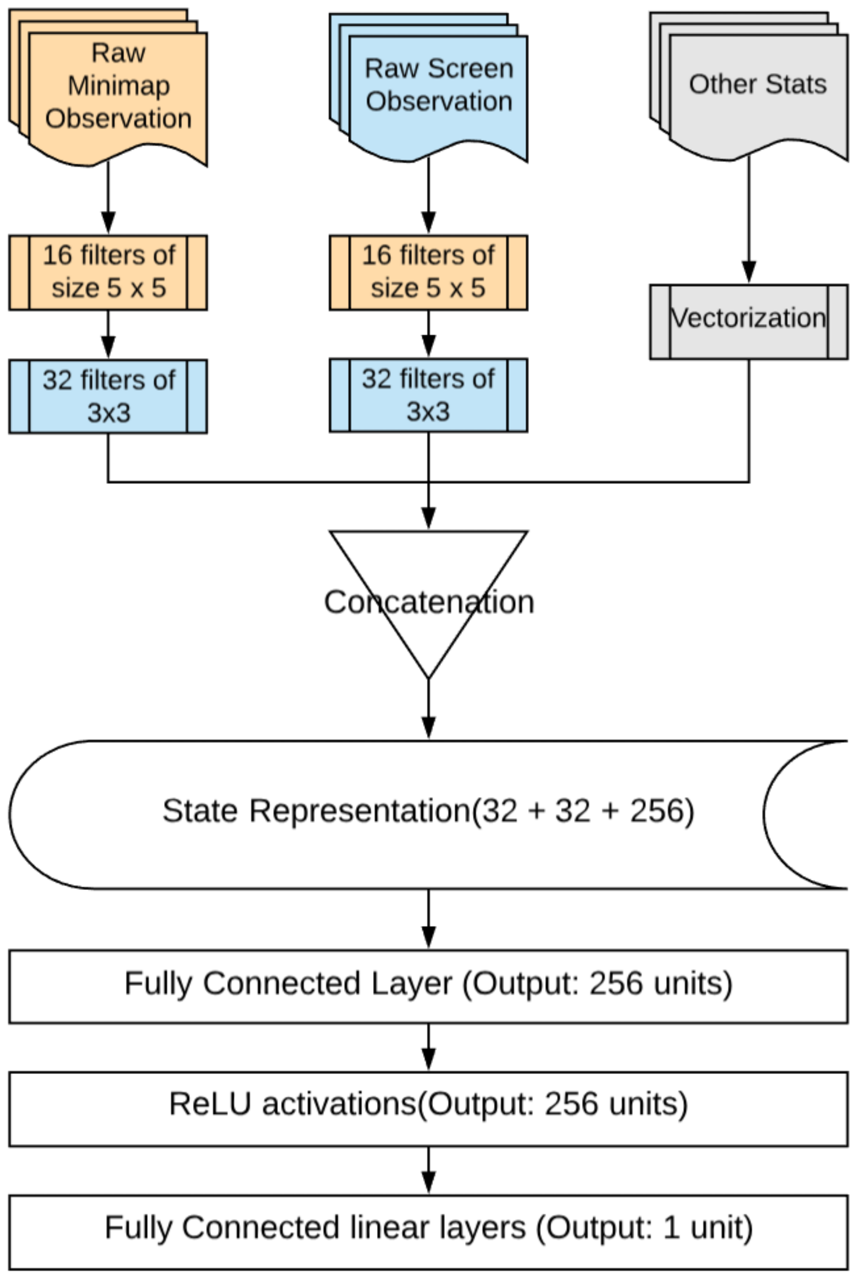

Figure 2.2: A detailed visual representation of the neural network's layers and structure.

2.2 Replicating DeepMind's Result

The trained neural network model should take a single observation at a time and predict the outcome of the game (win/lose).

2.2.1 Preparation

- The commonly found implementations are in TensorFlow. For simplification and clarity, the implementation is rewritten into Keras —- an abstraction Application Interface(API)[8]. The Keras uses TensorFlow back-end and hence is expected to produce similar results as direct TensorFlow implementation.
- The non-spatial data of every observation was fitted into a tensor using the Python package 'Numpy' in order to feed into the neural network.
- Due to limited computing resources, the observation number for each replay is reduced given the number of replays and GPU memory available.
- While an on-the-run data generator was created at the early stage of this research Figure 1.2, the response lag between the game server and the client game engine scaled up and significantly increased the training time of the model. Hence the observations were obtained before the training to reduce the training time as Figure 1.3.
- While the replay data set is enormous, a portion of it is not good for training because they are either:
  - too short (less than 1 minute of game duration) for analysis,
  - incorrect game version, or
  - corrupted replay

  A total of 6,440 bad replays were identified and the final available data set becomes 57,956 replays.
- The DeepMind's FullyConv model generated on Keras is shown in Figure 2.3

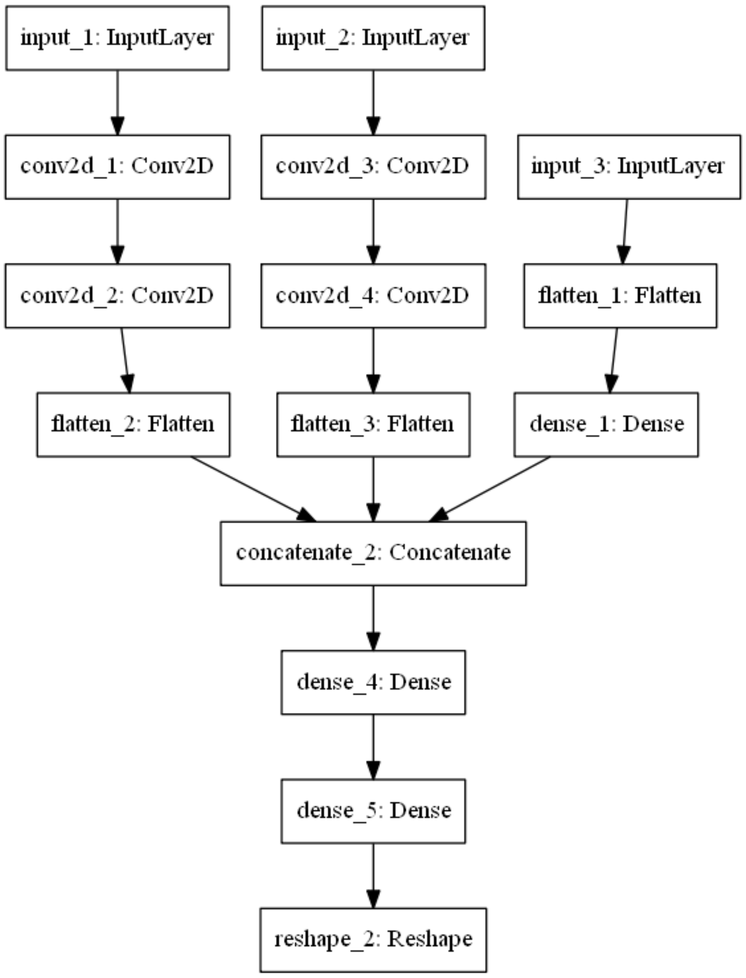

Figure 2.3: Keras's auto-generated network structure

- Due to the limited disk space and time for extracting replay data, only about 12K of the replays observations were obtained.

2.2.2 Training outcome

While the DeepMind claimed the accuracy could reach 65% with the network structure described above, trained on a well balanced 10K sub-data-set (win: lose = 50.05:49.95), the model's accuracy of prediction was at most 51% validation accuracy regardless of the time in the game (randomly sampled observation). Further training of the FullyConv network led to over-fitting and reduction in validation losses and accuracy. However, it should also be noted that this result may be caused by the significant difference in computing resources available for training. Since DeepMind has not yet released any source code or pre-trained model, this accuracy will be used to evaluate against the proposed model.

Chapter 3

Alternative Approach

3.1 Proposed Alternative

A recurrent neural network (RNN) model may fit better for this particular problem. RNN uniquely possesses the memory to comprehend past events by its ability to retain information. A few characteristics also implied that it is best solved using a model with temporal properties.

- **Balanced until major events**: An RNN network has memories which will have a higher chance to capture main events that affect the outcome than any standalone observation. For example, an observation taken in a surprise attack might make a trained standalone interpreter believe that the defender is at a disadvantage while a temporal RNN model might have captured the costly defense units that were built much earlier which could counter the attack easily. The two models would hence predict differently, with the latter one being more informed.

- **Chronological Ordering**: The inputs always comes in chronological order if there are more than one observations. It would be more natural as a solution to capture the ordering of the inputs since it is an important feature of the input data.

3.1.1 Problem Formulation

The problem can be formulated as the following to fit the RNN model.

Input : $\Sigma\{(\alpha_1, \beta_1, \gamma_1), (\alpha_2, \beta_2, \gamma_2), ..., (\alpha_20, \beta_20, \gamma_20)\}$
where α_t, β_t, and γ_t are the *t* th screen tensor (64x64x17), the *t*th minimap tensor(64x64x7), and the *t*th non-spatial vector (1x552) respectively from the extraction.

Output: The winner of the game (0/1) and the probability *p*.

In Deepmind's work, the model was trained based on 64 standalone observations taken at random from a single replay. However, for the proposed RNN network, the observations need to be arranged in chronological order, with this additional constraint, the model is ready to be trained.

3.2 Measure of Success

While the probability of the winning is substantial, the fact that the data-set came from a ranked-games makes the accuracy more important. Unlike professional league competition, in a ranked game, both players should have approximately similar skill, and hence the victory may only be achieved after the final combat of the game. This may not be captured by the random sampling method described in section 2. The goal of this model is to learn the accumulated advantages that hint the advantage in major events instead of the mistakes in battles (which out of the scope of this project).

Additionally, the probability of winning may fluctuate significantly depending on many factors which may not be captured by the simple model proposed here. Further study may be required in that case.

Therefore, the accuracy of the final prediction will be used as the primary measurement of the model's success in this paper.

3.3 Constructing the RNN model

In the DeepMind's work proposed a baseline LSTM structure, however, as suggested by Junyoung et al., GRU's outperforms LSTM most of the cases in practice[9]. Hence, this project will adopt GRU gates for performance sake. The prepossessing of the screen data and minimap data followed the original architecture in the FullyConv and had slight modifications to fit with the RNN and improve the model prediction accuracy. The network structure is shown below 3.1

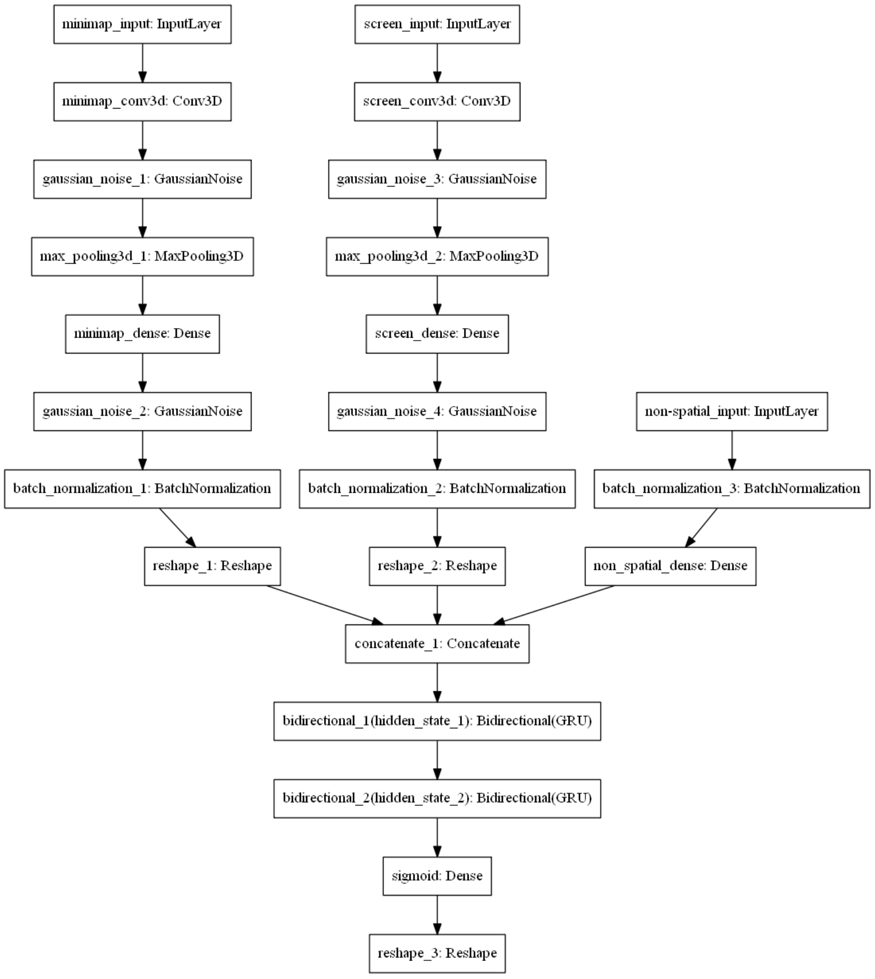
minimap_input: InputLayer
screen_input: InputLayer
minimap_conv3d: Conv3D
screen_conv3d: Conv3D
gaussian_noise_1: GaussianNoise
gaussian_noise_3: GaussianNoise
max_pooling3d_1: MaxPooling3D
max_pooling3d_2: MaxPooling3D
minimap_dense: Dense
screen_dense: Dense
gaussian_noise_2: GaussianNoise
gaussian_noise_4: GaussianNoise
non-spatial_input: InputLayer
batch_normalization_1: BatchNormalization
batch_normalization_2: BatchNormalization
batch_normalization_3: BatchNormalization
reshape_1: Reshape
reshape_2: Reshape
non_spatial_dense: Dense
concatenate_1: Concatenate
bidirectional_1(hidden_state_1): Bidirectional(GRU)
bidirectional_2(hidden_state_2): Bidirectional(GRU)
sigmoid: Dense
reshape_3: Reshape

Figure 3.1: An auto-generated visual representation of the network's flow by Keras.

3.4 Choices in the model

3.4.1 The Dying ReLU Problem

During preliminary training on the *Parameter Tuning* dataset 3.1, the model often stuck to a constant validation accuracy; we noticed that the predictions were either all win or all lose. Upon further investigations, we noticed that this was known as the "Dying ReLU Problem." Dying

ReLU problem refers to the cells that learned too many biases and thus causing the cells to become insensitive to any input. This is usually caused by inappropriate learning rate.

The dying ReLU problem is particularly tricky to handle with the limited computing power available. While one known solution is to lower the learning rate, the training time becomes too long to beat even the 50% accuracy. To avoid the dying ReLU problem, though not solving it, the activation functions were changed to 'tanh' to speed up the training. Finally, the Sigmoid activation and the binary-cross-entropy loss function were used to output the classification.

3.4.2 Optimizer

Many optimizers were tested with SGD being the fastest to learn without over-fitting as shown below. 3.2

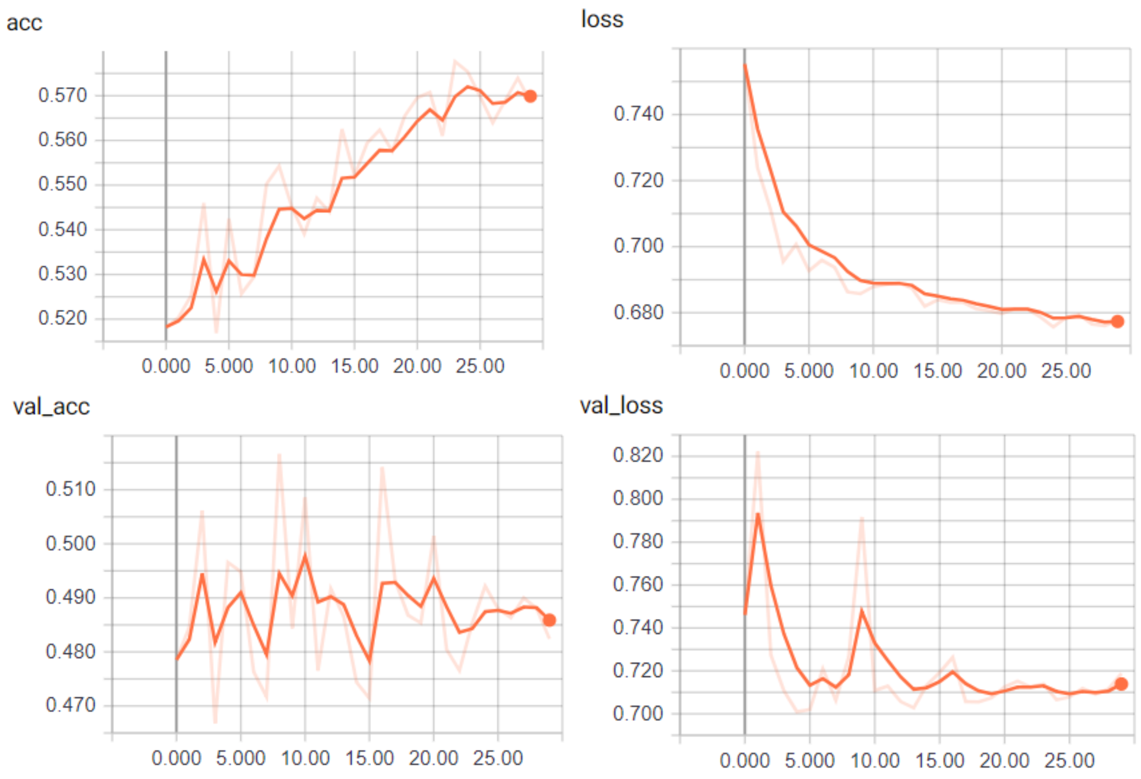

Figure 3.2: SGD optimizer with default parameters

3.4.3 GRU Layers

Additional Gaussian noise and dropout in the GRU layer were used to prevent over-fitting.

According to Keras documentation, the implementation of GRU was based on Cho's work [10] [11] and Gal's work [12] for dropout in particular.

3.5 Training the RNN model

The data set was partitioned into the following:

Table 3.1.

Function	**Size**
$Final\,Training$	$9K$
$Final\,Evaluation$	$1K$
$Test\,Training$	$1K$
$Parameter\,Tuning$	500
$Tuning\,Validation$	300
$Total$	$11.8K$

Table 3.1: Data-set Partition Details

The model was first trained on the 'Parameter Tuning' data-set and then evaluated on the 'Tuning Validation' validation data-set to allow fast parameter tuning. The following parameters were updated manually with respect to the small validation data-set:

Hyper-parameter	**Usage**
obs	Number of random observations from a replay (in chronoloical order)
$bSize$	the number of replays per batch
dr	the rate of dropout for the dropout layers and the RNN layers
lr	learning rate of the optimizer
epc	number of training epochs

Table 3.2: Data-set Partition Details

It is worth noting that during parameter tuning, the batch size and observation numbers both improve the accuracy in which the batch size is more significant. The observation number is reduced due to the limited memory of the training GPU. The ideal training parameter should be 64 or more observations and 64 replays per batch.

3.5.1 Configurations

Additionally, the mode was trained on two computers. One with GPU model GTX 1080 with 8GB of memory, CPU of i7-7700k@4.20GHz, RAM of 16GB, x64 Windows 10 operating system; and the other with GPU model GTX 1080 with 10GB of memory, Ubuntu 16 operating system. The TensorFlow version is 1.9.0 and Keras version is 2.2.2.

All codes are implemented in Python 3.7 except for the Keras data generator's fetch function. The fetch function was recompiled to Cython code to speed up the input assembly process. The final code and a pre-trained model with 58.5% accuracy can be found here:
https://github.com/lusensama/SC2-replays-learning

The project follows the workflow described below. 3.3 should provide a better overview of the project.

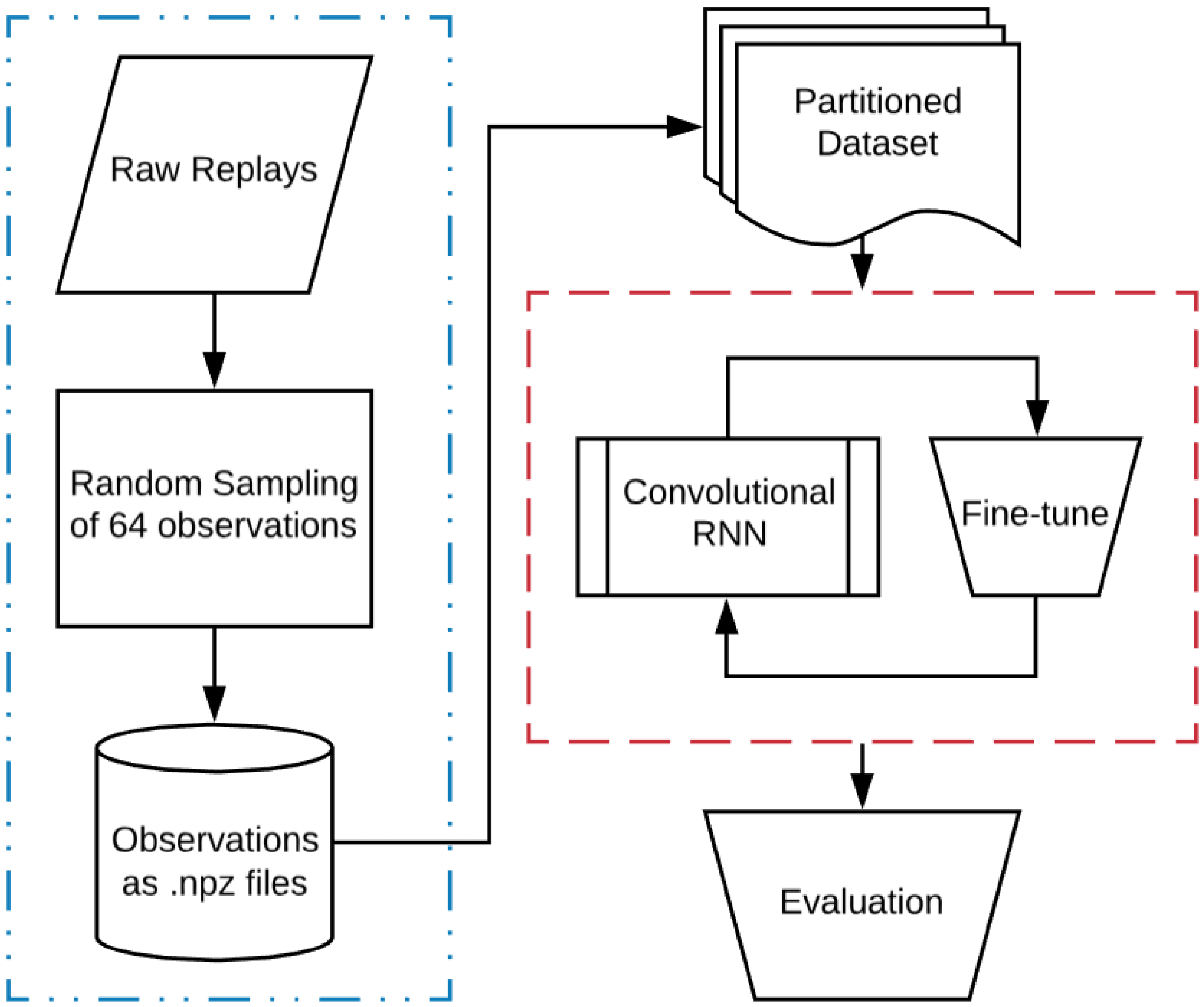

Figure 3.3: An overview of the project flow.

Chapter 4

Evaluation of Model

4.1 Performance

Figure 4.1 shows the final training results. Additional iterations of the training data lead to over-fitting, and hence epoch numbers were fixed to 10 after some trial-and-error.

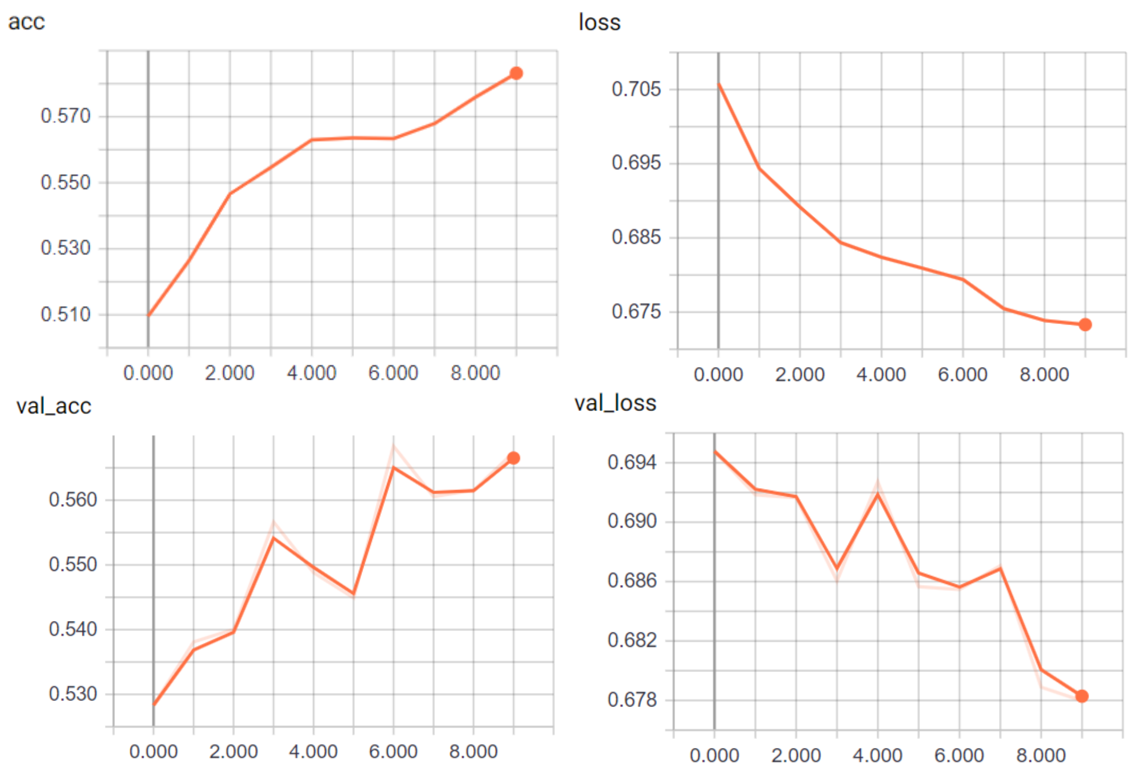

Figure 4.1: Final training

The training results for different partition of the data-set is as follow:

Data Set	Highest Validation Accuracy	Average Validation Accuracy
$Parameter\,Tuning$	0.5123	0.4971
$Test\,Training$	0.56	0.53
$Final\,Training$	0.5925	0.585

Table 4.1: Training with obs=16, $bSize$=64, dr=0.2, lr=0.0001, epc=10 15 (over-fitting was obvious on smaller data-set and hence reduced)

The training result is as following for the final model:

4.2 Ablation Study and Evaluation

4.2.1 Set Up

- **Base:** The base model takes all inputs, flattens and uses a Dense layer of one unit. The layer's activation function is 'sigmoid.'
- **Conv3D:** Two Con3D layers are added after both minimap tensor and screen tensor. The filter size=16, kernel size=(1,5,5), strides=1, activation='tanh.'
- **MaxPooling:** Two max-pooling layers are added after the Conv3D. The pool size is (1,2,2).
- **Dense/Fully Connected:** The Dense layers are added to all inputs. Both spatial inputs (minimap and screen) get 32 units, and the player stats get 256 units as output.
- **Gaussian Noise:** The Gaussian noise layers are inserted immediately after the inputs and after the Dense layer. The noise rate is 0.2 for all inputs.
- **Dropout:** The Dropout layers are inserted immediately after the inputs and after the Dense layer similar to Gaussian Noise layers but with the Gaussian noise layers removed. The dropout rate is 0.2 for all layers.
- **Bidirectional GRU:** This layer is added after the concatenation of all inputs processed thus far. The GRU layer has 128 units and does not return sequence (i.e., only returns final state).
- **Tuning:** The initializers, the number of observations, batch size, and learning rate of the SGD are tuned to increase the validation accuracy.

All the above mentioned network structure are compiled to SGD optimizer after final dense layer with 'sigmoid' activation similar to that of the Base network.

4.2.2 Result

The table below shows the accuracy of training the model for 4 epochs after stacking them in the following order. The accuracy is calculated from five runs.

Layer	Average Accuracy
Tuning	0.5453 (+0.0013)
Bidirectional GRU	0.5444 (+0.0261)
∗Dropout	0.5074 (+0.002)
GaussianNoise	0.5183 (+0.0129)
Dense	0.5054 (+0.0154)
Conv3D	0.4963 (+0.0182)
Base	0.4781

Table 4.2: Trained on $Final Training$ and validated on $Final Evaluation$ with default hyper-parameters, where Tuning uses the configurations described in section 4.1.
$*$ The Dropout layer replaced the Gaussian Noise layer instead of stacking on top.

4.2.3 Analysis

While it is common to use the Dropout layer, the Gaussian Noise layer performed better in the role of preventing over-fitting as shown by the average accuracy. The Dropout layer's accuracy was provided only for reference. It does not affect the final model's accuracy.

The changes in accuracy are the greatest after adding the Conv3D layers (+0.0182) and the GRU layer(+0.0261). Both layers' impacts are expected as they are the core of this model. The Conv3D layers interpret the spatial inputs while GRU learns the temporal relations of all features.

While the tuning seems to be ineffective, the number of epochs may be a potential factor affecting its impact.

4.2.4 Limitations

The accuracy of the final model is still low compared to that achieved by DeepMind's model. While the parameters can be further fine-tuned to improve the accuracy, we also did not fully exploit the data-set available due to the computing resources available. For example, the data size could be easily doubled by, for example, observing the opposing player's camera. Additionally, although the average was calculated, only 6 times were run in total due to the time constraint. Hence there might be an unexpected variance that made the accuracy appeared to be worse or better than records shown here. Lastly, the irregular sampling of the inputs of RNN is simplified and assumed as regular, because the regular sampling may require a more complex analysis of the player's behaviors and game property. This property may introduce potential problems that this project overlooks.

Chapter 5

Conclusion

5.1 Conclusion

While the final model is still less competent than the DeepMind's FullyConv model which achieved 65% accuracy, the locally trained FullyConv model using the same computing resources only barely beats the random guesser. However, on the other hand, the proposed alternative model achieved approximately 58% accuracy with an easily accessible GPU model GTX 1080. This result has proven the appropriateness of RNN as a potential solution to the game prediction problem.

5.2 Future Works

The limitations stated in the previous chapter should be addressed and may potentially increase accuracy. Naturally, the next step is to incorporate this model to the RL agent and serves as part of the scoring function input. Since the observations are all based on a single player's camera view, this should fit into the RL agent's framework with ease and thus improves the agent's capability overall.

Bibliography

[1] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[2] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft 2: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

[3] Kai Arulkumaran, Antoine Cully, and Julian Togelius. Alphastar: An evolutionary computation perspective. *arXiv preprint arXiv:1902.01724*, 2019.

[4] Peng Sun, Xinghai Sun, Lei Han, Jiechao Xiong, Qing Wang, Bo Li, Yang Zheng, Ji Liu, Yongsheng Liu, Han Liu, and Tong Zhang. Tstarbots: Defeating the cheating level builtin AI in starcraft II in the full game. *CoRR*, abs/1809.07193, 2018.

[5] Ho-Chul Cho, Kyung-Joong Kim, and Sung-Bae Cho. Replay-based strategy prediction and build order adaptation for starcraft ai bots. *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–7, 2013.

[6] Gabriel Synnaeve and Pierre Bessiere. A Bayesian Model for Opening Prediction in RTS Games with Application to StarCraft. In *Computational Intelligence and Games*, page 000, Seoul, South Korea, August 2011.

[7] Blizzard Entertainment Company. s2client-proto, 2014.

[8] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[9] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014.

[10] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoderdecoder for statistical machine translation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

[11] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoderdecoder approaches. *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, 2014.

[12] Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. In *Advances in neural information processing systems*, pages 1019–1027, 2016.

Sen Lu

EDUCATION

The Pennsylvania State University, University Park, PA — May 2019 (Expected)
Schreyer Honors College
Bachelor of Science, Computer Science
Minor in Japanese Language

RESEARCH EXPERIENCE

Research Lab Member — Summer 2017 – Fall 2018
Design Analysis Technology Advancement (DATA) Laboratory under *Dr. Conrad Tucker*
The Pennsylvania State University; University Park, PA

- Generated fake videos dataset for fake video detector based on given input audio
  - Trained pix2pix (Conditional Adversarial Nets) and mouth shape predictor (Recurrent Neural Network) synchronized
  - Built and managed docker environment for the DATA lab on DGX-1
- Trained classifier that detects fake videos using Inception Neural Network
  - Classified public forensic dataset at 87% accuracy
- Programmed 3DR SOLO Drone to track an object using computer vision on Android platform
  - Tracked the object based on color with almost no latency
  - Implemented the maneuver logic (in Kotlin) that keeps the drone at a similar speed of the moving object
  - Built application feature that allows the user to select the desired color for this application
  - Guided and supervised an REU junior who does a similar project with customized drone
- Created Bluetooth heart rate device communication app on the Android platform
  - Plotted a corresponding graph on the phone in real time
- Obtained facial biometrics through webcam videos in OpenCV
  - Implemented the Kalman Filter bounded tracking of a target face

Honor's Thesis Research — Fall 2018 – Present
Project: "Supervised Learning in RNN with SC2 Replays" supervised by *Dr. Daniel Kifer*
The Pennsylvania State University; University Park, PA

- Replicate DeepMind's Supervised Learning FullyConv network in Keras
- Trained a Recurrent Neural Network (LSTM) to predict the game outcome

REU Scholarship — Spring 2018
Project: "Parallel Run of OpenFOAM with External Program" with *Dr. Xiaofeng Liu* with
The Pennsylvania State University; University Park, PA

- Implemented parallel run of OpenFOAM program and external programs
  - Modified OpenFOAM C++ library to achieve the parallelism with an external program
  - Evaluated the methods and presented at the Undergraduate Research Exhibition Spring 2018
- Improved the effectiveness of existing documentation and tailored it for PSU students
  - Created a class inheritance diagram of OpenFOAM's frequently used classes
  - Wrote detailed OpenFOAM documentation aimed at researchers with no computing background only
  - Presented and lectured on OpenFOAM to *Dr. Liu*'s graduate students

Research Team Member — Fall 2016 – Spring 2017
Project: "Wearable Health" with *Dr. Sven Bilen*
The Pennsylvania State University; University Park, PA

- Assisted in building API for embedded system data processing
- Implemented Kalman Filter as a way of sensor raw data fusion

- Logged raw data from the sensors

PROGRAMMING LANGUAGES

Python, C++/C, Shell script(Linux), Java/Android, Kotlin, Docker, Verilog, LISP, LaTex

HIGHLIGHTED ADVANCED COURSEWORK

Machine Learning

- Machine Learning and Artificial Intelligence (CMPSC 448)
- Stochastic Modelling (Stats 416)
- Introduction to Machine Learning (IST 597 *Graduate level course*)
- Software Design Method – Senior Capstone Design (CMPSC 483)
- Machine Learning Applications in Forensics (IST 497)

Algorithms

- Data Structures and Algorithms (CMPSC 465) (CSE/BMMB 566 *Graduate level course*)
- Introduction to Computing Theory (CMPSC 464)
- Programming Language Concepts (CMPSC 461)

Computing System

- Operating System (CMPSC 473)
- Introduction to Computer Security (CMPSC 443)

PUBLIC PRESENTATIONS

- Presented at Undergraduate Research Exhibition at University Park, Penn State, Spring 2018 as a REU participant
- Presented at Undergraduate Research Exhibition at University Park, Penn State, Fall 2017 for Senior Capstone Design
- Presented as a CHOT (The Center for Health Organization Transformation) scholar in CHOT symposium Spring 2017

EXTRA CURRICULAR ACTIVITY

Tau Beta Pi Engineering Honors Society Webmaster Spring 2017 – Fall 2018

- Reconstruct official website from previous server
- Maintain website and update content