University of Waterloo
Faculty of Engineering
Department of Electrical and Computer Engineering

Integrating Heuristic Knowledge with Machine Learning: An Engineering Practice for Developing an AI that Plays StarCraft II

ECE498 Final Report

Group 2023.25

Prepared by:
Patrick Fourchalk (20755837), pgzfourc
Mengze Lyu (20755337), m6lyu
Enting Zhang (20755976), e35zhang
Ziqian (Carl) Fan (20727889), z45fan

Consultant: Prof. Douglas Harder
Submission Date: March 22, 2023

Abstract

In contemporary society, machine learning plays a significant role in solving scientific and real-world problems. However, there are many challenging problems that can not be easily solved by machine learning. For example, StarCraft II is considered a “grand challenge” for Artificial Intelligence research. The game’s non-transparent game state, real-time mechanics, and large action space result in difficulties in evaluating actions, making it impossible for machine learning algorithms to learn effectively. Therefore, this team developed an AI that plays the game StarCraft II as an engineering Practice example of how to integrate heuristic knowledge with machine learning. Extending the idea from AI in gaming, many more industries like military defense, energy distribution, and autonomous driving will benefit from the problem-solving approach of this project. Therefore, this project aims to set an example of how to pre-process an engineering problem into a high-level framework using heuristic knowledge and then integrate it with modern deep learning models. As for this specific engineering problem, the AI has to determine its own way of building troops, deploying tactics, and maximizing combat performance to increase the winning rate. With human heuristic knowledge, this team’s design approach is to divide the problem into ”build order”, ”scouting”, and ”combat” sub-problems as well as utilize existing machine learning models along with business logic. Our project, which integrates a heuristic framework and machine learning components, is expected to outperform the existing AI AlphaStar in terms of the training complexity and feasibility of learning transfer given a small change in the problem (a game balance/feature change patch).

Acknowledgements

First, we would like to acknowledge our project advisor Douglas Harder for overseeing our project and learning about the game StarCraft II to accurately evaluate and question the design of our project.

Second, we would like to thank the StarCraft II AI community for sharing prior research and open-sourcing many engineering approaches. Specifically, bot author & API manager BurnySc2 for maintaining the python API (python-sc2 open-sourced library) over Blizzard Entertainment’s provided C++ StarCraft II game API; bot author & Sharpy Framework major contributor DrInfy for establishing and open-sourcing a StarCraft II AI development template (sharpy-sc2 open-sourced library).

Third, we would like to thank all the professors in the ECE department who taught us knowledge related to software engineering and machine learning. You introduced us to this challenging yet opportunity-rich field of study.

Table of Contents

List of Figures

List of Tables

1 High-Level Description of Project

1.1 Motivation

In contemporary society, machine learning plays a significant role in solving scientific and real-world problems. Because of the game's non-transparent game state, real-time mechanisms, and huge action area, StarCraft II is seen as a "grand challenge" for AI research [1]. As a result, such a challenging problem can not be easily solved by machine learning. For example, AlphaStar, an AI published by Google that used mostly human-supervised learning and only a small portion of reinforcement learning to play this game, demonstrates the fact that even a technological empire that has the most talented programmers and a vast amount of resources cannot solve some problems effectively by only using machine learning. Also, AlphaStar's approach relies heavily on resources that small companies can not afford, which undermines the ability to apply its engineering practice to the industry. Furthermore, AlphaStar's approach fails to demonstrate interpretability in its decision-making, which is a significant disadvantage given that many AI problems require that the AI's decisions should be certain and should make sense to a human to satisfy ethical and security concerns [2]. As a result, our engineering approach to implementing this AI, which can both solve complex problems sufficiently well with limited resources and preserve interpretability, would serve as an engineering practice for the industry. Extending the idea from AI in gaming, many more industries like military defense, energy distribution, and autonomous driving will benefit from the problem-solving approach of this project.

1.2 Project Objective

The objective is to design an artificial intelligence that can determine its own way of building troops, deploying tactics, and maximizing combat performance to increase the winning rate in the game StarCraft II. Simultaneously, the AI should show the interpretability of the decision-making process throughout the game.

1.3 Block Diagram

The proposed solution is an engineering practice example of a heuristic framework integrated with some machine learning components. The software architecture design diagram is shown in **Figure 1**. As the block diagram shows, we will implement the Sharpy-sc2 Framework, Scouting Manager, Build Order Manager, and Unit Control Manager subsystems.

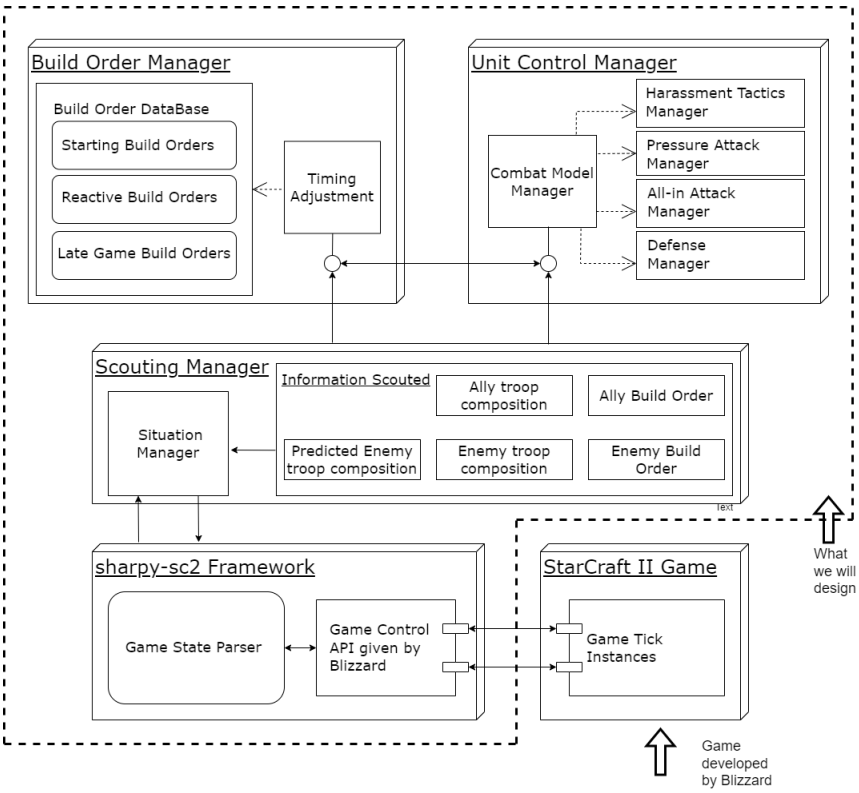

Figure 1: Block Diagram for this project. Note that because the project is an AI that plays StarCraft II, both the input and output are API calls that interact with a game host server.

2 Project Specifications

2.1 Functional Specifications

Table 1 outlines the functional specifications of our project.

Specification	Subsystem	Description
FS1: Game State Parsing	sharpy-sc2 Framework	The Game State Parser must be able to read from Game Control API, parse the data into three kind and send to each manager: the Scouting Manager, Unit Control Manager and Build Order Manager.
FS2: Situation awareness handling	Scouting Manager	The situation manager must be able to judge which player has advantage in current situation and send the judgement to the Builder Order Manager and the Unit Control Manager.
FS3: Combat Managing: Arrange Attacks	Unit Control Manager	The combat module manager must be able to choose a reasonable attack strategy based on the situation.
FS4: Harassment Tactics Execution	Unit Control Manager	The harassment tactics manager must be able to command a few troops to harass enemy bases and scout enemy build order.
FS5: Pressure Attack Execution	Unit Control Manager	The pressure attack manager must be able to command a powerful army (based on gameplay and time) to destroy enemy's newest base or prevent enemy from expanding bases.
FS6: All-in Attack Execution	Unit Control Manager	The all-in attack manager must be able to produce as many troops as possible and command them for a decisive battle.
FS7: Asynchronous attacks	Unit Control Manager	The combat module manager should be able manage and trigger multiple combat actions at the same time.
FS8: Enemy attacks defensing	Unit Control Manager	The defense manager must be able to produce and command troops to defend enemy's attacks.
FS9: Build order controlling	Build Order Manager	The timing adjustment must be able to manage the AI player's build order.

Table 1: Functional Specifications

2.2 Non-functional Specifications

Table 2 outlines the non-functional specifications.

Specification	**Subsystem**	**Description**
NFS1: Security	IP White-List Manager	The system should be able to run only on local server to prevent people use it on Ranked games against other players.
NFS2: Compatibility	Entire Project	The system should be able to run on all computers that can run the game StarCraft II properly.

Table 2: Non-functional Specifications

3 Detailed Design & Prototype Data

3.1 Sharpy-sc2 Framework

This subsystem is responsible for:
1. Set up the AI vs AI or AI vs human game environment for local development,
2. Wrap up the Blizzard StarCraft II gameplay API in Python,
3. Package the AI into a zip file for online AI ladder competitions.

3.1.1 Environment Setup

This subsystem will communicate with the Blizzard Game API, which launches a game instance. The Python code that initiates a match amongst bots within the project directory is shown below. This will be the only legal approach to construct this component in order to directly interact with the server while playing the game. Alternatively, we would need to create a robot to interface with the physical keyboard and mouse which would be difficult given our time constraints.

```
def add_definitions(definitions: BotDefinitions):
    definitions.add_bot(
        "protossbot",
        lambda params: Bot(
            Race.Protoss,
            ProtossBot(BotDefinitions.index_check(params, 0, "default"))
            ),
        None
    )
    definitions.add_bot(
        "terranbot", lambda params: Bot(Race.Terran, TerranBot()), None
    )
    definitions.add_bot(
        "zergbot", lambda params: Bot(Race.Zerg, ZergBot()),
        None
    )

def main():
    root_dir = os.path.dirname(os.path.abspath(__file__))
    ladder_bots_path = os.path.join("Bots")
    ladder_bots_path = os.path.join(root_dir, ladder_bots_path)
    definitions: BotDefinitions = BotDefinitions(ladder_bots_path)
    add_definitions(definitions)
    starter = GameStarter(definitions)
    starter.play()
```

3.1.2 Game Play API Wrapper

This subsystem will encapsulate the Blizzard Game Tick data structures in higher-level logic, allowing for faster AI development. "Build another base" is a nice example, which is constructing a new base to collect more resources at a different site on the map and expand your area of control. This is a high-level action that players are familiar with; however, from the perspective of a machine that can only call the Blizzard APIs, there are numerous little atomic activities involved. First, iterate through all of the available sites for a base. Second, dispatch a worker to the desired place. Third, determine whether we have enough funds to construct a base. Finally, instruct the worker to utilize the "building a base" ability at the spot. Also, there might be edge cases where our enemy is trying to block that location so we can not issue the "build" ability. We have to handle these situations without throwing errors. Therefore, the subsystem will abstract the higher-level logic functions to summarize the low-level actions(because the machine learning component should not care too much about these low-level details and focus more on decision-making). Another role that this component must perform is to summarise the information into a more logical structure for usage by AI developers. For example, the game only tells the AI which units are present and what properties they have in a massive List data structure. However, using more human-readable logic, it does not distinguish between friendly and enemy units. As a result, the component must be capable of subdividing the entire game state information into

subsets of relevant logical information. To accomplish this, this component will be designed entirely around "business logic," with numerous manager classes being built to parse the game state into understandable forms. Below is an image that covers all the managers in the Game Play API Wrapper Subsystem. They are the core component of the existing Sharpy framework and our designs will be built on top of it with overridden logic.

Name	Date modified	Type	Size
__pycache__	2021-10-09 9:42 PM	File folder	
grids	2021-10-09 9:42 PM	File folder	
roles	2021-10-09 9:42 PM	File folder	
__init__.py	2021-10-09 9:41 PM	Python File	1 KB
act_manager.py	2021-10-09 9:41 PM	Python File	2 KB
action_manager.py	2021-10-09 9:41 PM	Python File	9 KB
building_solver.py	2021-10-09 9:41 PM	Python File	31 KB
cooldown_manager.py	2021-10-09 9:41 PM	Python File	4 KB
enemy_units_manager.py	2021-10-09 9:41 PM	Python File	9 KB
gather_point_solver.py	2021-10-09 9:41 PM	Python File	3 KB
income_calculator.py	2021-10-09 9:41 PM	Python File	3 KB
log_manager.py	2021-10-09 9:41 PM	Python File	3 KB
lostunitsmanager.py	2021-10-09 9:41 PM	Python File	4 KB
manager_base.py	2021-10-09 9:41 PM	Python File	1 KB
pathing_manager.py	2021-10-09 9:41 PM	Python File	15 KB
previousunitsmanager.py	2021-10-09 9:41 PM	Python File	2 KB
unit_cache_manager.py	2021-10-09 9:41 PM	Python File	8 KB
unit_role_manager.py	2021-10-09 9:41 PM	Python File	12 KB
unit_value.py	2021-10-09 9:41 PM	Python File	40 KB
unit_value_test.py	2021-10-09 9:41 PM	Python File	2 KB
version_manager.py	2021-10-09 9:41 PM	Python File	10 KB
zone_manager.py	2021-10-09 9:41 PM	Python File	28 KB

Figure 2: Managers in the Game Play API Wrapper Subsystem

Figure 3: Some important managers will show their states during the game

3.1.3 Packager

There is an AI competition ladder "SC2 AI Arena" developed by a global StarCraft II game AI development community. Researchers can develop their AI and deploy them on the server to compete with others' AIs. To package our AI into a form that can participate in the competition, we need this subsystem to package the code. This can be done by using the Python code below.

```
def main():
    zip_keys = list(zip_types.keys())
    parser = argparse.ArgumentParser(
        description="Create a Ladder Manager ready zip archive for
        SC2 AI, AI Arena, Probots, ..."
    )
    parser.add_argument("-n", "--name", help=f"Bot name: {zip_keys}.")
    parser.add_argument("-e", "--exe", help="Also make executable
    (Requires pyinstaller)", action="store_true")
    args = parser.parse_args()
    bot_name = args.name
    if bot_name == "all" or not bot_name:
        zip_keys.remove("all")
        for key in zip_keys:
            zip_types.get(key).create_ladder_zip(args.exe)
    else:
        if bot_name not in zip_keys:
            raise ValueError(f"Unknown bot: {bot_name},
            allowed values are: {zip_keys}")
        zip_types.get(bot_name).create_ladder_zip(args.exe)
```

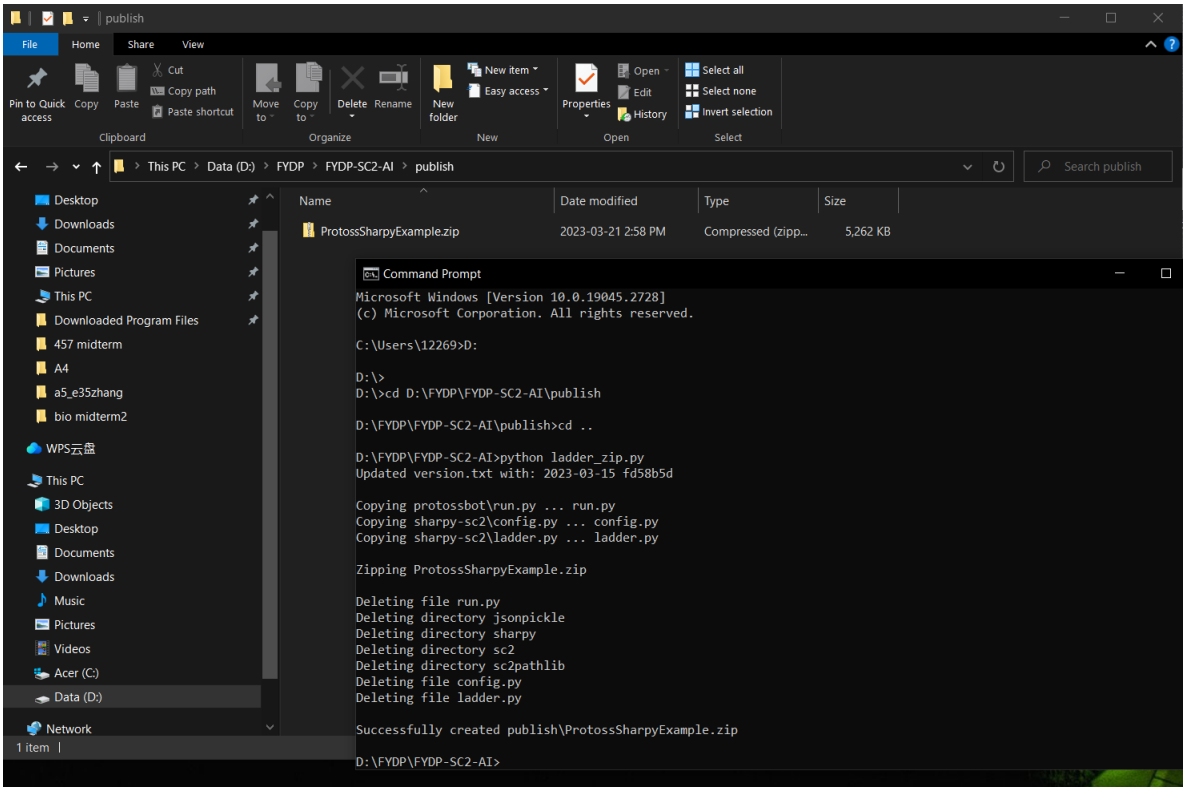

Figure 4: This is an example of how to publish a bot

3.1.4 Local Development

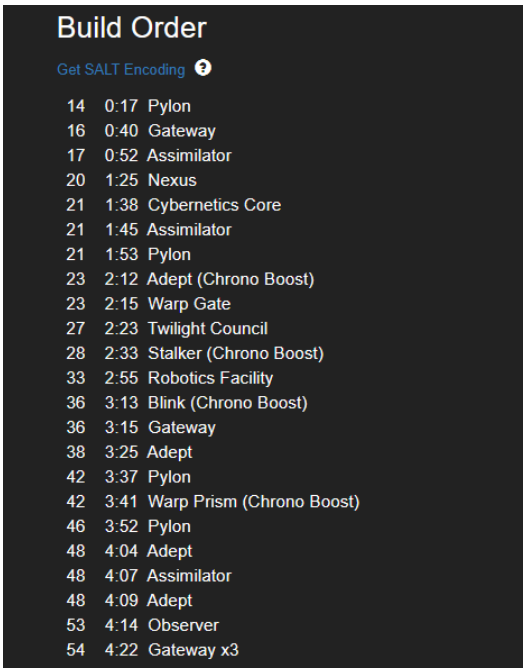

Figure 5: This is an example of build order in online game guides

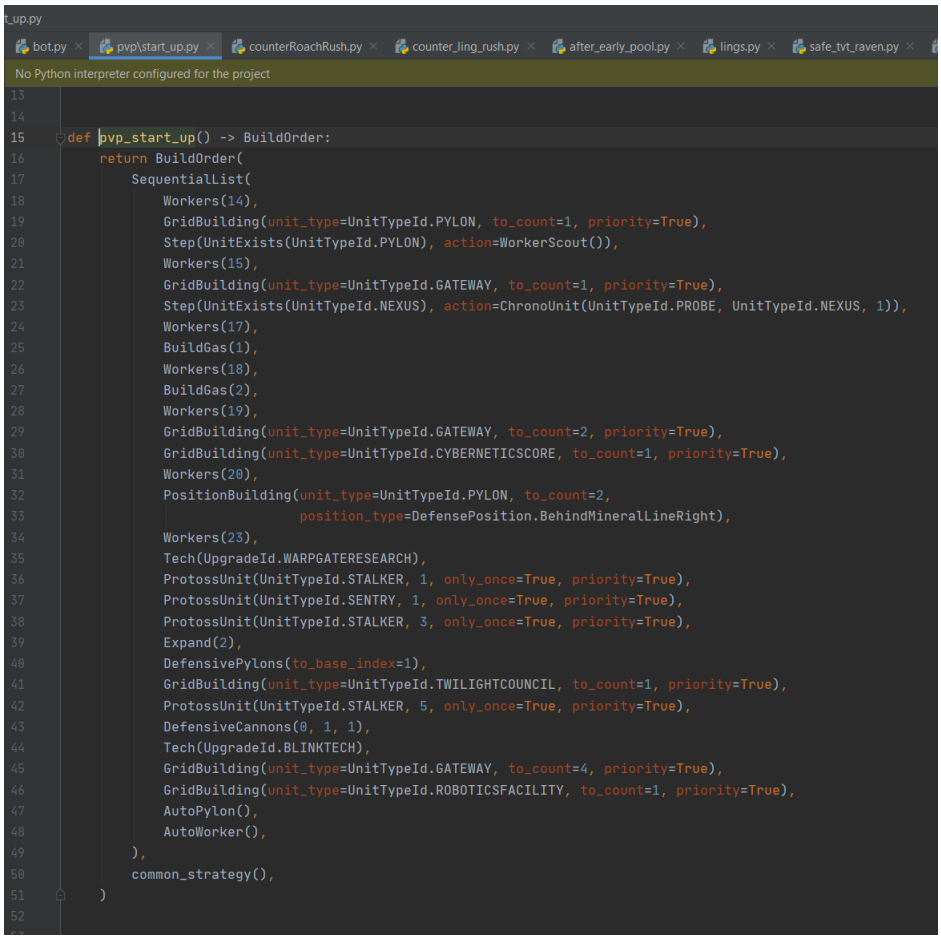

Figure 6: This is an example of how to code a build order using Sharpy

As we can see, our code represents a very high level of abstract logic and allows developers to rapidly develop functional code.

3.1.5 Conclusion of Sharpy-sc2 Framework

This component ensures that the AI will directly interact with the server while playing the game. Our final design decision is to implement this component with pure "business logic" Python code and not include any machine learning components. The specific designs reflect the human heuristics of this game. From the screenshots, we can conclude that this component sufficiently satisfies the requirements of "The Game State Parser must be able to read from Game Control API, parse the data into three kinds and send to each manager: the Scouting Manager, Unit Control Manager, and Build Order Manager."

3.2 Scouting Manager

StarCraft II has non-transparent game states, which means if there is no map visibility, a fog of war prevents our AI from observing what the enemy is doing. Our AI will periodically send units to enemy bases and use the unit's vision to reveal the opponent's build order, unit combination, and enemy territory. To implement the scouting manager, we will have a class called "Knowledge" to record all the information found:

```
class Knowledge:
    def __init__(self):
        ...
        self.zone_manager: ZoneManager = ZoneManager()
        self.enemy_units_manager:
            EnemyUnitsManager = EnemyUnitsManager()
        self.build_detector: BuildDetector = BuildDetector()
        self.enemy_army_predicter = EnemyArmyPredicter()
        ...
```

3.2.1 Zone Manager

We need a zone manager to record which zones are controlled by our AI player and the enemy. This manager will help us with "map control". The manager will focus on zones controlled by each player, and then our AI will be able to distinguish enemy-controlled areas from ally-controlled areas. Meanwhile, the manager keeps track of the expanded zones of each player, zones which we have not scouted yet, and zones viable for expansion. This information helps the scouting manager recognize the enemy's strategy and expanding ally bases. Below is example code for the zone manager:

```
    def likely_enemy_start_location(self) -> Optional[Point2]:
        return self.zone_manager.expansion_zones[-1].center_location

    def enemy_start_location_found(self) -> bool:
        return self.zone_manager.enemy_start_location_found

    def unscouted_zones(self) -> List[Zone]:
        unscouted = [z for z in self.zone_manager.all_zones
            if not z.is_scouted_at_least_once]
        return unscouted

    def expansion_zones(self) -> List[Zone]:
        return self.zone_manager.expansion_zones

    def enemy_expansion_zones(self) -> List[Zone]:
        return self.zone_manager.enemy_expansion_zones

    def our_zones(self) -> List[Zone]:
        ours = [z for z in self.zone_manager.all_zones if z.is_ours]
        return ours

    def our_zones_with_minerals(self) -> List[Zone]:
        filtered = filter(
            lambda z: z.our_townhall and z.has_minerals, self.our_zones)
        return list(filtered)

    def own_main_zone(self) -> Zone:
        return self.zone_manager.own_main_zone

    def enemy_main_zone(self) -> Zone:
        return self.zone_manager.enemy_main_zone
```

3.2.2 Enemy Units Manager

In order to see through the enemy strategy, we need an enemy units manager that records the number of each enemy unit. The value will be updated every time an ally unit is sent for "scouting". There are three main unit types need to be recorded: workers, structures and troops.
The number of workers represents the economic strength of a player, and can provide information about what strategies the enemy might be planning to use. For example, if the enemy's worker count stays below 22 (which is the maximum worker capacity for one base), then it is likely that they are attempting a one-base rush tactic, which sacrifices late economic and technological advantages for an early army strength advantage. The structures and troops reveal the "build order" of a player, which is the significant part of their strategy. If we can keep an eye on the enemy's build order, we will be able to actively maintain countermeasures against the enemy.

This is example code in the Enemy Units Manager that counts the number of enemy workers and checks the enemy's troop composition:

```
def unit_types(self) -> KeysView[UnitTypeId]:
    return self._known_enemy_units_dict.keys()

def enemy_worker_count(self) -> int:
    worker_type = self.knowledge.enemy_worker_type
    return self.unit_count(worker_type)

def enemy_composition(self) -> List[UnitCount]:
    lst: List[UnitCount] = []
    for unit_type in self._known_enemy_units_dict:
        unit_count = self.unit_count(unit_type)

        if unit_count > 0:
            lst.append(UnitCount(unit_type, unit_count))

    return lst
```

3.2.3 Build Detector

After the Scouting Manager updates the enemy unit and structure information each time we scout the enemy, in order to be reactive, our AI needs to recognize the enemy build order and select a counter measure to execute. Thus, a build detector is required. The build detector will categorize the enemy's build order into three types: Economic builds, Timing Attack builds and Rush builds. Also, the build detector will use the enemy's build order to determine the appropriate counter build order to execute. Our main job is to characterize each build order and train the AI to recognize them. Supervised learning can be used here with timing, structure orders, and troop compositions as training parameters, but decided to use reinforcement learning first because it is easier for us to tell the AI whether it performs well in the environment rather than attempt to tune a large number of parameters that determine how the AI functions. However, when we translate the game domain, we discovered that the game domain is huge and that the agent cannot converge to a fixed result within our estimated time frame. In the end, we decided to build a decision tree using a finite state machine. Instead of using AI to explore the enemy's build order, we hard coded all the potential proper build orders that then enemy could have based on the number of enemy units and buildings found through scouting, and let our agent search for the best counter measure using the decision tree. NN is applied for classification, and we generated training data by creating dummy AI bots that plays each build orders we coded into our decision tree.

At the start of each game, both players begin with the same preset: 12 workers and 1 main base, and this will be the starting point of our decision tree(figure 7). After we sent some units and scout the enemy's build order being aggressive, we will update this and try finding a counter measure through our Build Order Manager. In figure 8, our bot scout that the enemy is using a build order called "12 pool", which is a early rush builder, then it will search for a counter build order and apply it.

Figure 7: The enemy's builder always begin with Start

Figure 8: The bot scout the enemy's build order and update it

If we had a more peaceful mid-game with the enemy, as the game processes, our bot will move on into a late game build order. This only happens when we developed good economy and solid army composition for defence since we will invest a lot in production building and technology.

Figure 9: Enter late game after a peaceful mid-game

3.2.4 In-game Economic and Army analyser

In order for our bot to recognize the situation in the game, we designed an in-game economic and army analyser. Since each worker has the same mining efficiency, we can predict the enemy's income based on its worker number. The number of bases also influences the enemy's economy as each base has a maximum capacity of active workers. Meanwhile, the size of enemy's army is scouted and predicted based on the scouted enemy's build order and income level. This help our bot to plan timing attacks. According to different build order, the criteria of a good attack timing can be different, but generally, when key technology research is finished or we have clear advantage in army against our enemy, we should start an attack to maintain the lead in this game.

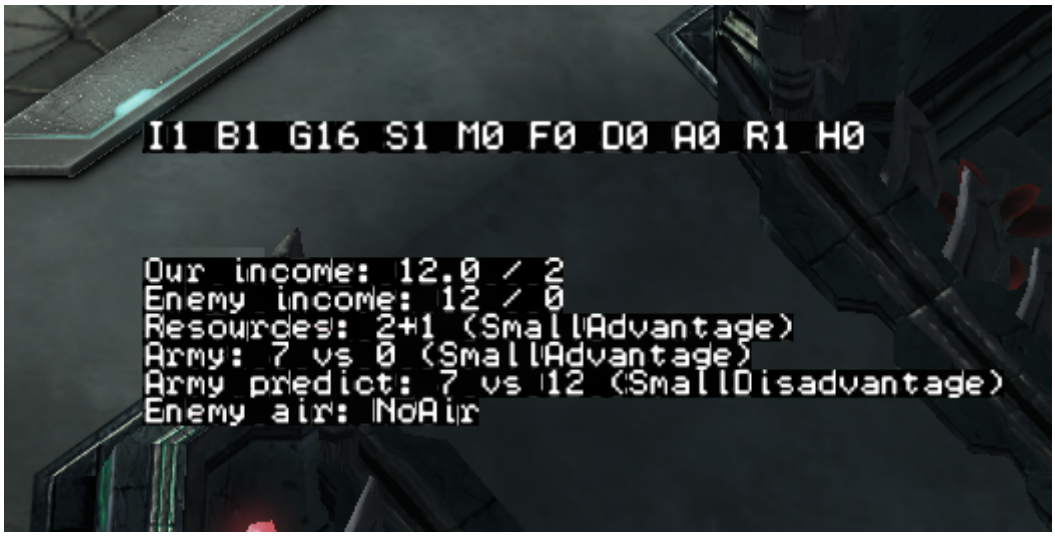

Figure 10: the analyser is updated continuously

3.2.5 Scouting Manager Specific Design Decisions

When designing the agent that will be used in our scouting manager, there are several choices: create a new neural network, apply the reinforcement learning methods, or use a pre-trained network in our project. In figure 3, the behaviour of a pre-trained network and a newly trained network is shown.

Figure 3(a) shows the win rate difference between training a pre-trained agent and training from scratch when curriculum learning is applied. Both methods have poor performance after 250 iterations, since the win rate does not rise above 50%. Meanwhile, the win rate of the agent that trained from scratch is significantly low - it is lower than 10% after all iterations. Thus, the agent in this project should be pre-trained to save training time and computational resources.

Figure 3(b) shows the difference in win rate when module training is applied. It is obvious that the pre-trained agent achieves a much higher win rate than the agent trained from scratch. It also shows that the module training has a higher win rate than those when curriculum learning is applied, so applying a pre-trained network will have better performance in this project.

When considering applying neural network to our project, several problems were raised to be solved. First, a StarCraft II game usually ends between 7-9 miniutes, thus training the NN with game records will take huge amount of time that we cannot afford. Second, the game domain is too big for us to convert it into python that can create an environment for the agent to explore. All the units in StarCraft II can move freely and both player will control over 50 units in a combat. If no limit is applied, the agent will not converge even in 1000 epochs. So we decided to restrain the game domain to proper size and combine it to our decision tree. And we take advantage from StarCraft2 AI community that we applied a pre-trained neural network, which takes in the parsed game data, classify the game stage and enemy strategy, then output immediate build order by searching the decision tree.

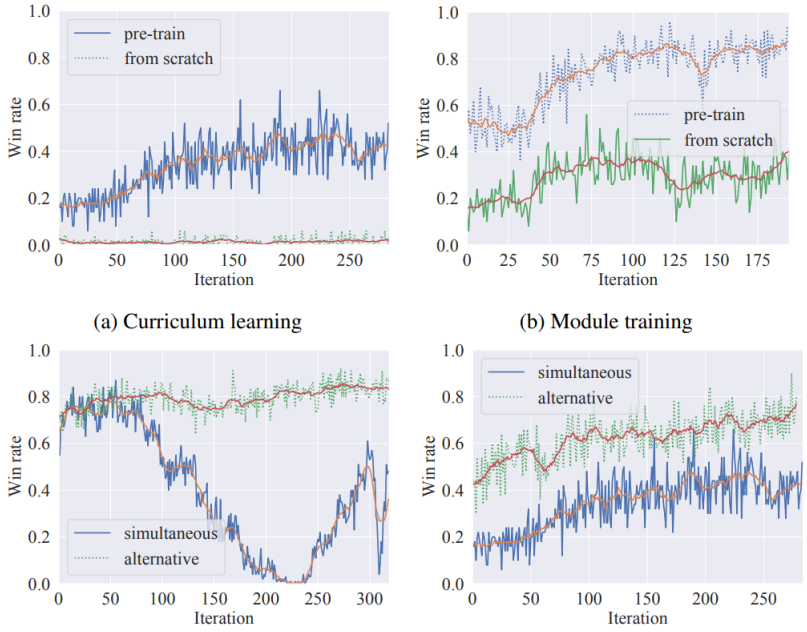

Figure 3: Winning curve in training process.

Figure 11: Pre-Trained Learning vs Trained from scratch Learning[8]

In order to improve the interpretability of the AI, supervised learning will be used in examining human replay data, especially those data from expert players. It will be used to train the agent to make decisions in different stages of the game, especially for the mid game build order trade offs and army compositions. It shows that the AI's decision making is more like human that it might use different strategies against different enemies and it is confident in late game combats instead of escaping fights and decision making by "cheeseing". The performance of the AI will be compared with various players, starting from scripted bots to human players. It can be expected that the AI can give interpretative strategies learnt from its opponents.

3.3 Build Order Manager

Build order is one of the most important components in StarCraft II. It defines the building procedures or process in the game, and by implementing a good build order, your army in the game will be well engaged. By well engaged, we mean that the roles/armies the player uses are able to build more economic and unit production structures and equipped with better weapon upgrades, which it is described as "Macro". Better "Macro" means higher "purchasing power" in the game, which will lead to better game infrastructure and a larger-scale army.
Our team is aiming to design a build order database with three components as it is shown in the block diagram above. They are namely "Starting Build Orders", "Reactive Build Orders" and "Late Game Build Orders". The following code uses Python lambda function to realise the build order switches according to Enum provided by build detector.

```
def pvp_main_force(self) -> BuildOrder:
    return BuildOrder(
        Step(lambda k: k.build_detector.rush_build ==
            EnemyRushBuild.Macro, self.pvp_eco_start_up()),

        #  pvp
        Step(lambda k: k.build_detector.rush_build ==
            EnemyRushBuild.SafeExpand, self.pvp_micro()),

        Step(lambda k: k.build_detector.rush_build ==
            EnemyRushBuild.WorkerRush, self.
            counter_ProxyZealots()),
        Step(lambda k: k.build_detector.rush_build ==
            EnemyRushBuild.ProxyZealots, self.
            counter_ProxyZealots()),

        Step(lambda k: k.build_detector.rush_build ==
            EnemyRushBuild.CannonRush, self.counter_CannonRush
            ()),

        Step(lambda k: k.build_detector.rush_build ==
            EnemyRushBuild.ProxyBase, self.counter_4BG()),
        Step(lambda k: k.build_detector.rush_build ==
            EnemyRushBuild.Zealots, self.counter_4BG()),
        Step(lambda k: k.build_detector.rush_build ==
            EnemyRushBuild.AdeptRush, self.counter_4BG()),

        Step(lambda k: k.build_detector.rush_build ==
            EnemyRushBuild.AirOneBase, self.counter_air()),

        Step(lambda k: k.build_detector.rush_build ==
            EnemyRushBuild.ProxyRobo, self.counter_Robo()),
        Step(lambda k: k.build_detector.rush_build ==
            EnemyRushBuild.RoboRush, self.counter_Robo()),

        Step(lambda k: k.build_detector.rush_build ==
            EnemyRushBuild.EarlyExpand, self.
            counter_EarlyExpand()),

        Step(lambda k: k.build_detector.rush_build ==
            EnemyRushBuild.FastDT, self.counter_FastDT()),
    )
```

These lines of code can be found in "bot.py" file under the "build_order" folder. The file "bot.py" is the decision tree of the build order segment. It works as a finite state machine, and only one parameter would be sent to ".rush_build".

It is worth mentioning that the program will first detect what race we are countering to (i.e., Protoss, Zerg, or Terran). Then, the following code will assign the corresponding lambda func-

tion set (i.e., the different build order to different race). Note that the build order for the Terran race would be the same as the build order to the Protoss race.

```
def __init__(self, build_name: str = "default"):
    super().__init__("FYDP")
    self.conceded = False
    self.builds: Dict[str, Callable[[], BuildOrder]] = {
        "pvp": lambda: self.pvp_build(),
        "pvz": lambda: self.pvz_build(),
        "pvt": lambda: self.pvt_build(),
        "pvr": lambda: self.pvr_build()
    }
    self.build_name = build_name

def configure_managers(self) -> Optional[List[ManagerBase]]:
    return [BuildDetector()]

async def create_plan(self) -> BuildOrder:
    if self.build_name == "default":
        if self.knowledge.enemy_race == Race.Protoss:
            self.build_name = "pvp"
        elif self.knowledge.enemy_race == Race.Zerg:
            self.build_name = "pvz"
        elif self.knowledge.enemy_race == Race.Terran:
            self.build_name = "pvt"
        else:
            self.build_name = self.build_name = "pvr"

    self.data_manager.set_build(self.build_name)
    return self.builds[self.build_name]()
```

3.3.1 Starting Build Order

"Starting build order" is the number one priority of the gaming procedure the A.I. will use to start the game. From the previous code snippet, this line of code send the information of the starting build order:

```
Step(lambda k: k.build_detector.rush_build == EnemyRushBuild.
   Start, pvp_start_up())
```

As it is shown in the figure below (Figure 12), this is the top right corner of the game interface, and it clearly marked that enemy's build order is also "Start", which makes sense because it was the very beginning of the game. That means our troop is deploying the starting build order as well.

Figure 12: Top right corner game screen snapshot

There are possibly many different kinds of starting build orders. Our team will train A.I. to have the ability of deploying the starting like "Quick-Attack Starting" and "Economy-Priority

Starting", etc. (Other types of starting will be integrated when our team or A.I. discover new tactics.) "Quick-Attack Starting" is obvious by its name. The A.I. will quickly deploy troops and form an army to attack the enemy quickly. This is a strategy to diminish enemy's army's possibility of growing larger, and thus, enemy's odds are "strangled in the cradle". This strategy is also useful in engineering design, and for example, when the power grid is turned on, it is designed to satisfy all possible customer's needs, so how to quickly start a smart grid for customers is an important task for A.I. However, the risk of deploying "Quick-Attack Starting" is that if some troops in enemy's team have better economy, it is more likely those troops will dominate the battle. As for "Economy-Priority Starting", as it is discussed above, the A.I. decides to first develop its ability for building more towers, training more advanced army, and buying more equipment. This is a strategy for protracted battle. Besides, A.I. will assign corresponding tactics when it discovers how the enemy is operating in a protracted war. This leads to the next block of "Reactive Build Orders".

3.3.2 Reactive Build Order

As the game goes, if A.I. detects the build order of the enemy, the A.I. would probably deploy "Reactive Build Orders". It is a build order that being reactive and changing the strategy in the battle. For instance, in the mid to late game phase, if our side is developing the "economy" while the A.I. suddenly detects that enemy is deploying the "Quick-Attack" tactic, the A.I. will determine if it is sufficient to change the strategy to defense enemy's intensive "Quick-Attack". It is obviously that during the battle, "Reactive Build Orders" will be called repeatedly, since a good battle shall also take other's strategy into account and make changes accordingly. Referring back to the smart grid example above, customers' demand will not stay unchanged, especially nowadays there are increasingly more electric cars need to charge. If the smart grid detects that customers' demand behaviors are changing, the A.I. will decide how to efficiently allocate the needs and avoid power overload or even power outage.

```
def counterRoachRush() -> BuildOrder:
    AutoPylon(),
    AutoWorker(),
    ChronoUnit(UnitTypeId.IMMORTAL, UnitTypeId.
       ROBOTICSFACILITY, 2),
    ChronoUnit(UnitTypeId.WARPPRISM, UnitTypeId.
       ROBOTICSFACILITY, 2),
    DoubleAdeptScout(2),

    Step(EnemyUnitExists(UnitTypeId.ZERGLING, 4),
     ProtossUnit(UnitTypeId.ADEPT, priority=True, to_count=1))
        ,
    Step(EnemyUnitExists(UnitTypeId.ZERGLING, 8),
     ProtossUnit(UnitTypeId.ADEPT, priority=True, to_count=6))
        ,

    SequentialList(
        BuildGas(2),
        GridBuilding(unit_type=UnitTypeId.GATEWAY, to_count=2,
            priority=True),
        GridBuilding(unit_type=UnitTypeId.CYBERNETICSCORE,
           to_count=1, priority=True),
        GridBuilding(unit_type=UnitTypeId.PYLON, to_count=2,
           priority=True),
        ProtossUnit(UnitTypeId.ZEALOT, priority=True, to_count
           =1, only_once=True),
        Step(UnitExists(UnitTypeId.PYLON, 2, include_not_ready
           =False),
            GridBuilding(unit_type=UnitTypeId.ROBOTICSFACILITY
               , to_count=1, priority=True)),

        BuildOrder(
```

```
                ProtossUnit(UnitTypeId.IMMORTAL, priority=True,
                    to_count=1),
                ProtossUnit(UnitTypeId.STALKER, priority=True,
                    to_count=4, only_once=True),
            ),

            BuildOrder(
                ProtossUnit(UnitTypeId.WARPPRISM, priority=True,
                    to_count=1),
                ProtossUnit(UnitTypeId.STALKER, priority=True,
                    to_count=4),
                Tech(UpgradeId.WARPGATERESEARCH),
                ProtossUnit(UnitTypeId.ADEPT, priority=True),
                ProtossUnit(UnitTypeId.IMMORTAL, priority=True),
            ),
    )
```

This above code is the build order for countering the "Roach Rush" build order from the enemy. The scouting manager sends the information that enemy is doing "Roach Rush", and in that way, the program will initiate this build order.

There is also a sequential list that is worth mentioning from the same file (i.e., counterRoachRush.py):

```
def common_strategy() -> SequentialList:
    return SequentialList(

        OracleHarass(),
        DistributeWorkers(),
        PlanHallucination(),
        HallucinatedPhoenixScout(),
        PlanCancelBuilding(),
        WorkerRallyPoint(),
        PlanZoneGather(),
        PlanZoneDefense(),
        Step(UnitExists(UnitTypeId.ZEALOT), action=
            DoubleAdeptScout()),
        Step(UnitExists(UnitTypeId.WARPPRISM), action=
            PlanZoneAttack()),
        PlanFinishEnemy(),
    )
```

These lines of codes are easy to understand by its nature. However, one need to notice that it is different from the finite state machine (or lambda functions) discussed previously. The sequential list constraints the execution of each line of code, and they have to be executed in order, or as the name says, in sequence.

3.3.3 Late Game Build Order

As we enter the end phase of the game, A.I. will deploy "Late Game Build Orders". The A.I. will start to produce the most advanced troops with this build order. There are not too many strategies in the "Late Game Build Order" because the army arrangements or managements are similar for different army races in the late game. Besides, if the A.I. detects the probability of winning the battle is very low, it will deploy the build order that how to end the game quickly to save time and boost game efficiency.

3.3.4 Specific Designs regarding Build Order

It is worth mentioning that "Timing Adjustment" also plays a significant role in the "Build Order Manager" according to the block diagram. It is independent of the three build orders explained above. Those three build orders form a "build order database", and the timing adjustment will recognize and decide which phase the player is at during this battle (i.e., starting, mid-phase, and late phase) based on the time progression.

The manager network described in the block diagram works together when A.I. is participating the battle. The build order manager will obtain the enemy's current strategy and situation through scouting manager. "Reactive Build Orders" only knows how to counteract after the scouting manager detects enemy's situation and sends that information to build order manager. Besides, every time a build order is enabled, a series of unit controls will also be enabled, from "expand base" to "train troops". No manager is independent of each other, and the A.I. shall use all three managers well together.

3.4 Unit Control Manager

"Unit Control" is a crucial aspect of the game player's ability, and can greatly influence the outcome of a match. For example, a ranged unit can win a fight without taking damage against a melee unit if it keeps "kiting", a tactic where the ranged unit moves around to make the grounded enemy chase it and thus not be able to attack, or take lots of damage if it stands still while attacking. Therefore, we design a Unit Control Manager to execute harass, pressure, all-in attack and defend tactics. Before units can be controlled, the A.I. must decide what tactics to use in its strategy, which is determined in the Build Order Manager.

3.4.1 Combat Model Manager

To specifically control friendly units during combat, there are seven types of moves that encompass how the Unit Control Manager can command a unit or group of units while moving to or stationed at a location:

- Search and Destroy: unit(s) search far distances and hunt enemy units or structures to attack and destroy.
- Assault: unit(s) move to the target location or unit to attack it, and will attack any enemies they come across while moving. Corresponds to the "attack" and "attack-move" command in Star Craft II.
- Push: unit(s) move slowly towards an enemy in a defensive formation, fortifying their next position before advancing. Will fight back against enemy attacks but will continue to move towards their target.
- Defensive Retreat: unit(s) retreat back to their base while shooting at the enemy if they are being pursued, which slows the retreat. Useful for protecting other retreating units or to destroy weakened enemy units before fully disengaging.
- Panic Retreat: unit(s) retreat from battle as quickly as possible to avoid any more casualties, using all available resources and abilities to escape.
- Harassment: unit(s) ignore targeting enemy buildings and army units and target enemy workers to disrupt the opponent's strategy. These unit(s) should be kept alive for harassment to be successful.
- Regroup: unit(s) move towards a nearby group of friendly units to regroup.

Assigning these move types to specific units or, more commonly, groups of units during battle changes what actions the A.I. will command each unit to do. For example, units that are retreating will "move" to their target location which ignores all enemy units, while units the are pushing will "attack-move" to their target which allows them to engage any enemies they encounter on the way to their target location:

```
if (move_type == MoveType.DefensiveRetreat or
        move_type == MoveType.PanicRetreat
    ):
    combat.move_to(group, target, move_type)
    break
...
elif move_type == MoveType.Push:
    # Don't worry about closest enemies if we are pushing
    combat.attack_to(group, target, move_type)
```

3.4.2 Combat Behaviour Design

How the AI decides what action should be assigned to each unit depends on the current state of the game. It compares certain statistics of the current game state against thresholds that our group designs to choose what move type is the best at that time. For most of our business logic, we use thresholds that optimize the performance of the A.I. when it is controlling units using the built-in information about the game. For example, we already know that the Protoss unit Stalker has an attack range of six, therefore we tell the bot to never attempt to attack an enemy that is outside of this attack range. However, there are some thresholds which require the A.I.

to evaluate a subjective situation and make a decision on what to do, such as when to retreat from a fight:

```
if enemy_local_power.is_enough_for(own_local_power, multiplier *
    self.retreat_multiplier):
     # Start retreat next turn
     self.print(
         f"Retreat started at {own_local_power.power:.2f} own local
             power "
         f"against {enemy_local_power.power:.2f} enemy local power
             ."
     )
     return AttackStatus.Retreat
```

How do we decide what the values of these fuzzy thresholds should be? When designing the unit control manager, we first had to use pre-trained values for these because our goal at that time was to verify that our other managers were working as intended. The pre-trained values also give our bot a good starting point where it can control the units to some extent, but the values are not optimized for the build orders that are used in the Build Order Manager. To optimize many of these values, we decided to use a type of meta-heuristic algorithm because from what we learned in ECE 457A, these types of algorithms are designed to be generic search algorithms that explore the search space and slowly reduces the search space and exploits the best solutions to find a good solution. Also, three out of four of our group members have completed ECE 457A so it is more efficient for us to implement and understand. Out of the possible meta-heuristic algorithms, two were considered by our team: genetic algorithm and genetic programming.

Genetic algorithm allows us to use the parameters we want to tune as our genes for each member of our population in an array that is easy to represent in Python, and explores the entire search space at once. StarCraft II also has a built-in score system, which is useful for evaluating the performance of each bot during its evolution. Genetic programming allows us to modularize each individual action the bot can perform and use the same genetic algorithm to find the best combinations of actions for the bot to execute from any situation; however, it is very complex to implement dynamic code that can accomplish this, and professional StarCraft II players have already found methods to handle most situations in the game which we can teach to the bot without machine learning. These factors swayed us to choose to use genetic algorithm to optimize our parameters.

In our first design of the genetic algorithm, we used a population size of 10, a whole arithmetic crossover rate of 0.6, a uniform mutation rate of 0.1, and an alpha value of 0.75. The population was chosen such that there is a balance between genetic diversity and realistic run-times of the algorithm, since each game of StarCraft II averages about 10 minutes in length, so if each bot plays 3 games in the population, then each generation would take approximately 300 minutes 5 hours to complete. Even training for 20 generations is a heavy toll on any of our computers, so this is our maximum computational capacity. The other values were chosen based on a ECE 457A assignment which was based on finding the real-valued genetic algorithm parameters that converged on the global maximum in the least number of generations. However, we placed too many parameters that we wanted to optimize to represent the genetics of each bot in the population. This led to an issue that is encountered often in the field of machine learning and that we later learned about in ECE 457B, the curse of dimensionality. When you have too many features you are trying to optimize at the same time, the learning algorithm tends to diverge and does not perform well. In our case, our parameters converged the same way as if we were choosing which bot performed the best at random, which can be seen in the similarity between the following two graphs:

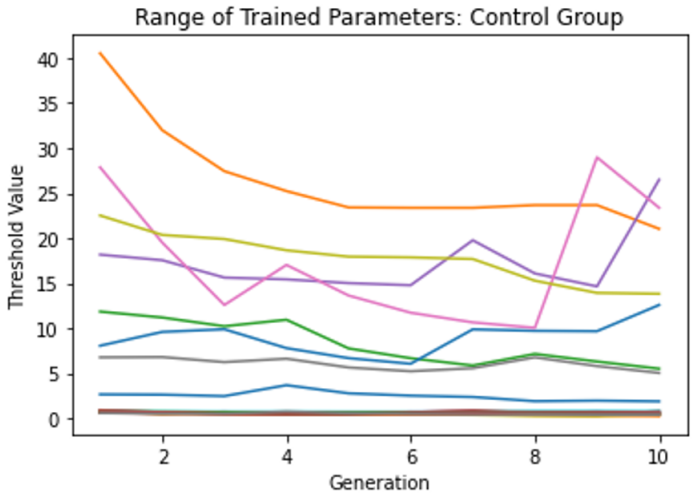

Figure 13: Graph showing random convergence of 19 of our parameters simultaneously. The bot chosen to move to the next generation is random.

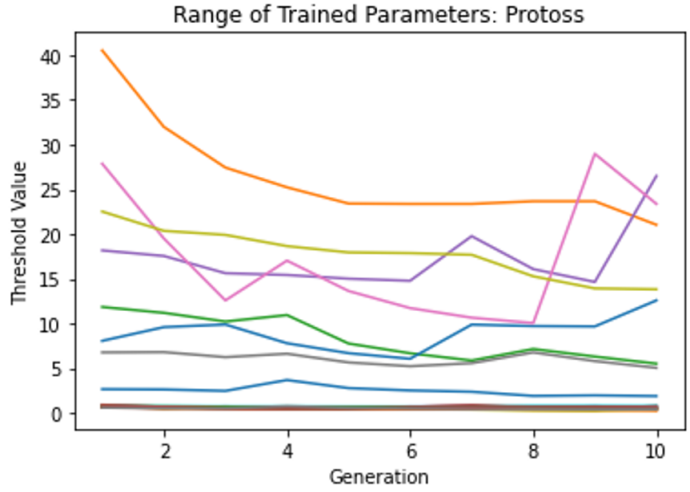

Figure 14: Graph showing the bot's parameter convergence during the genetic algorithm. The bot with the highest average score after 3 games is chosen to move to the next generation.

Our second and more successful design of the genetic algorithm trains only a few parameters at a time, so over many iterations of this we were able to obtain better parameters for the Unit Control Manager to achieve higher scores on average. The score of the game is calculated by how long the game took, who won the game, how many friendly units or structures were lost in battle, and how many enemy units or structures were destroyed over the course of a match.

In our third and final design, we split the training of the parameters to be different when the bot is fighting against each of the different races in the game to allow the bot to find more optimal values against each race rather than the best average parameters for any opponent. This behaviour allows the A.I. to meet the function requirement described in FS3, since it is able to dynamically change the strategy based on the current accessible information. The bot stores the final configuration of the tuned parameters so they can be accessed by the Unit Control Manager during run-time.

```
class PlanZoneDefense(ActBase):
    def __init__(self):
        super().__init__()
        self.config = None
            if os.path.exists(CONFIG_FILE):
```

```
            with open(CONFIG_FILE, 'r') as f:
                self.config = json.load(f)

async def worker_defence(
    self, defenders: float, defense_required, enemy_center,
       zone: "Zone", zone_tags, zone_worker_defenders
):
    if self.config is not None and self.knowledge.enemy_race
       is not None:
        enemy_race = race_to_str(self.knowledge.enemy_race)
        unit_health_defense_threshold = self.config[enemy_race
           ][UNIT_HP_DEF_THRESHOLD]
        ...
    else:
        # Default to pre-trained values
        unit_health_defense_threshold = 0.6
        ...
```

Our consultant did suggest the alternative of using fuzzy control logic instead of using fixed threshold values to determine the unit control actions of the bot, but by that point we had already started the development of our current design and we had no knowledge of how to implement a fuzzy control system until this term in the course ECE 457B, which left little time to redesign our crisp business logic approach.

3.4.3 Unit Control Manager Subsystems

There are four major attack types that the Combat Manager controls using subsystems:

- Pressure Attack Manager: A timed attack that takes advantage of a perceived strength difference between the friendly and enemy armies. The enemy should be pressured to spend resources to defend their bases, and retreating is a viable option. Before a pressure attack is executed, the A.I. compares its economy and army strength to what it knows about the enemy's position and may abandon attacking if the enemy is deemed to be in an advantageous position. Choosing which enemy base to target is currently done by the following function, which is designed to target an enemy base within the proximity of friendly territory. This is done to meet FS5:

```
    def _get_target(self) -> Optional[Point2]:
        our_main = self.zone_manager.expansion_zones[0].
           center_location
        proxy_buildings = self.ai.enemy_structures.closer_than
           (70, our_main)

        if proxy_buildings.exists:
            return proxy_buildings.closest_to(our_main).
               position

        # Select expansion to attack.
        # Enemy main zone should the last element in
           expansion_zones.
        enemy_zones = list(filter(lambda z: z.is_enemys, self.
           zone_manager.expansion_zones))

        best_zone = None
        best_score = 100000
        start_position = self.gather_point_solver.gather_point
        if self.roles.attacking_units:
            start_position = self.roles.attacking_units.center

        for zone in enemy_zones:  # type: Zone
            not_like_points = zone.center_location.distance_to
               (start_position)
```

```
                not_like_points += zone.enemy_static_power.power *
                    5
                if not_like_points < best_score:
                    best_zone = zone
                    best_score = not_like_points

        if best_zone is not None:
            return best_zone.center_location
```

- All-In Attack Manager: A large-scale attack that requires trading your economy for immediate military strength. This strategy is performed similarly to a pressured attack, except the A.I. will never retreat in an attempt to maximize the damage done to the enemy.
- Harassment Tactics Manager: Sending a singular or small group of units to target enemy workers instead of the army or buildings. This disrupts the enemy's economy and diverts the enemy player's attention away from their current plan. In the following harassment function, the Protoss unit Adept is harassing the enemy by using its teleport ability to infiltrate the enemy base to gather information:

```
    async def execute(self) -> bool:
        if self.build_detector and self.build_detector.rush_detected:
            self.roles.clear_tasks(self.scout_tags)
            self.scout_tags.clear()
            return True  # Never block

        if not self.zone_manager.enemy_start_location_found:
            # We don't know where to go just yet
            return True  # Never block

        if self.ended:
            return True  # Never block

        if not self.started:
            await self.check_start()

        if self.started:
            adepts: Units = Units([], self.ai)
            for tag in self.scout_tags:
                adept: Unit = self.cache.by_tag(tag)
                if adept is not None:
                    adepts.append(adept)
            if len(adepts) == 0:
                await self.end_scout()
            else:
                await self.micro_adepts(adepts)

        return True  # Never block
```

  The enemy's buildings, constructions, and army composition will allow the A.I. to narrow down their build order, which satisfies FS4 for this project.

- Defense Manager: If there are enemy units near a friendly base, friendly units must be mobilized to defend that base from damage. The defend functionality is able to detect the enemy presence in friendly bases, if there are enough friendly army units to defend against it, and assign units to defend or produce more defenders during an attack:

```
if zone_defenders.exists or
    zone.is_ours or zone == self.zone_manager.own_main_zone
):
    if not self.defense_required(enemies):
        # Delay before removing defenses in case we just
        # lost visibility of the enemies
        if (
```

```
                zone.last_scouted_center == self.knowledge.ai.time or (
                    self.zone_seen_enemy[i] + PlanZoneDefense.ZONE_CLEAR_TIMEOUT
                    < self.ai.time
                )
            ):
                self.roles.clear_tasks(zone_defenders_all)
                zone_defenders.clear()
                zone_tags.clear()
                continue  # Zone is well under control.
        else:
            self.zone_seen_enemy[i] = self.ai.time

        # Identify enemy location
        if enemies.exists:
            # enemy_center = zone.assaulting_enemies.center
            enemy_center = enemies.closest_to(zone.center_location).position
        elif zone.assaulting_enemies:
            enemy_center = zone.assaulting_enemies.closest_to(
                zone.center_location
            ).position
        else:
            enemy_center = zone.gather_point

        defense_required = ExtendedPower(self.unit_values)
        defense_required.add_power(zone.assaulting_enemy_power)
        defense_required.multiply(1.5)

        defenders = ExtendedPower(self.unit_values)

        for unit in zone_defenders:
            self.combat.add_unit(unit)
            defenders.add_unit(unit)

        # Add units to defenders that are being warped in.
        for unit in self.roles.units(UnitTask.Idle).not_ready:
            if unit.distance_to(zone.center_location) < zone.radius:
                # unit is idle in the zone, add to defenders
                self.combat.add_unit(unit)
                self.roles.set_task(UnitTask.Defending, unit)
                zone_tags.append(unit.tag)

        if not defenders.is_enough_for(defense_required):
            defense_required.substract_power(defenders)
            for unit in self.roles.get_defenders(
                defense_required, zone.center_location
            ):
                if unit.distance_to(zone.center_location) < zone.radius:
                    # Only count units that are close as defenders
                    defenders.add_unit(unit)

                self.roles.set_task(UnitTask.Defending, unit)
                self.combat.add_unit(unit)
                zone_tags.append(unit.tag)

        if len(enemies) > 1 or (
            len(enemies) == 1
            and enemies[0].type_id not in UnitValue.worker_types
        ):
    # Pull workers to defend only and only if the enemy isn't one worker scout
            if defenders.is_enough_for(defense_required):
                # Workers should return to mining.
```

```
                for unit in zone_worker_defenders:
                    self.roles.clear_task(unit)

                    zone.go_mine(unit)
                    # Just in case, should be in zone tags always.
                    if unit.tag in zone_tags:
                        zone_tags.remove(unit.tag)
                # Zone is well under control without worker defense.
            else:
                await self.worker_defence(
                    defenders.power,
                    defense_required,
                    enemy_center,
                    zone,
                    zone_tags,
                    zone_worker_defenders
                )
```

Since the Defense Manager is able to evaluate the strength of the attacking enemy and takes appropriate action to defend against the attack successfully, it meets our FS8 requirement.

Figure 15: An in-game image demonstrates zealot unit control creating a wall blocker against an incoming attack from the left.

Figure 16: An in-game image demonstrates an all-in attack of our units (blue/green) against the enemy.

3.5 Integration

"A competent player can always multitask on macroing build orders, controlling the units, and maintaining situation awareness," stated Xudong Huang, a professional Star Craft II player. [7] Thus the four major components should be able to execute simultaneously in our design. This can be achieved by Python's asynchronous programming. Because the Unit Control Manager do not consume any in-game resources and only instructs units' actions and Build Order Manager only performs building constructions and unit training, there will be no race conditions between the two. Same thing goes for Build Order Manager and scouting Manager. But there are certain scenarios where race conditions will occur between Scouting Manager and Unit Control Manager. Scouting is achieved by sending units to explore enemy territories. If a unit is performing scouting and the Unit Control Manager wants it to perform combat tactics, the two managers will control the unit at the same time and send conflicting commands. Our design solution is to introduce another "Unit Role Manager" subsystem in the Unit Control Manager. The Unit Role Manager will register a role to any allied units. When a unit is marked as "Scouting", the Unit Control Manager will not have access to control that unit. If an urgent situation occurs, the Scouting Manager will signal the "Unit Role Manager" that we need all the units in battle stations. Then the unit will be marked as "combat" or "defense" which the Unit Control Manager will have access to.
As demonstrated above, all the subsystems will be connected according to the Block Diagram in Figure 1. This can ensure the cooperation among subsystems and maximize the change of wining a game.

4 Discussion and Conclusions

4.1 Evaluation of Final Design

The current design meets the functional requirements for this problem sufficiently well and has a relatively small complexity to implement(finished by a group of four), making it a good example for our proposed "Integrating heuristic knowledge and machine learning components" engineering approach. However, our proposed design still has its limitations.

In the current design, the logic and parameters used by the A.I. to decide how to build troops, deploy tactics, and maximize combat performance are all deterministic as well as realized using the human-programmed heuristic framework. This reflects the design philosophy of "human heuristic knowledge accelerates machine learning" yet introduces more human error into the project. The current design can be successfully implemented because StarCraft II players have already found some optimal strategies through their years of gaming experience, so building an A.I. with weightings already determined by humans saves time and computing power to gather training data and train a recursive neural network. We can not guarantee such success for other engineering problems that do not have a relatively large prior human heuristic knowledge base.

4.2 Use of Advanced Knowledge

This project involves many API calls to the StarCraft II game server instances to perform in-game actions, and those calls all happen asynchronously. Upper-year knowledge from ECE 350: Real-time Operating Systems were used to handle concurrent executions of tasks. Knowledge from ECE 457A: Cooperative and Adaptive Algorithms, ECE 457B: Computational Intelligence, and some design philosophy from ECE 457C: Reinforcement Learning have been used during this project as well. There are many Software engineering problems that need to be solved when developing such an AI. The heuristic framework component of this project alone requires more than 10,000 lines of Python coding. Therefore, knowledge of managing such a huge code repo and developing highly-abstracted logic have been critical so far, which all came from advanced ECE Software-Engineering-related courses. ECE 495: Autonomous Vehicles provide the fundamental engineering philosophy to approach the problem(divide and conquer each sub-problem using different modern machine learning or traditional algorithms) Therefore, this project is considered using a lot of the Upper-year advanced knowledge.

4.3 Creativity, Novelty, Elegance

StarCraft II is such a complex game that even the problem formulation is hard. This project successfully parses this game into a quantitative form and applies a gaming control library in Python, allowing advanced machine-learning strategies to be implemented easily.

Unlike AlphaStar which fails to demonstrate interpretability in its decision-making, our AI is an integration of machine learning and conventional business logic programming, providing interfaces to demonstrate why decisions were made. This project can be considered an example of how to design a machine learning project where interpretability and certainty are crucial to satisfy ethical and security concerns.

Unlike AlphaStar which requires vast resources to train and implement (Google even cooperated with Blizzard to acquire data from bot vs human games), our engineering approach only takes four fourth-year university students with around 1,000 hours of effort in total.

Therefore, we conclude that our solution involves a great amount of Creativity, Novelty, and Elegance.

4.4 Quality of Risk Assessment

Given the fact that our project is purely software-based, it doesn't have any physical risks. However, this AI can be used by people for cheating purposes in human ladder games. This is why we implemented an IP whitelist functionality. It will detect if the bot was connected to a simulation game server or a Blizzard-hosted server. This prevents the bot to be run against human players on the ladder to ruin other's game experiences and at the same time preserves human vs AI functionality when running in a non-Blizzard-hosted environment. Before the symposium day on May 15th, the AI was confirmed to not function in a Blizzard-hosted server.

All in all, we evaluated and tested all the risks of the project and it was a success on the symposium day.

4.5 Student Workload

Group Member	**Work Done (%)**
Mengze Lyu	26
Ziqian (Carl) Fan	24
Enting Zhang	26
Patrick Fourchalk	24

Table 3: Percentage of total work done by each group member.

5 References

[1] R. Ali, T. Daniel, B. David ..., "AlphaStar: Mastering the real-time strategy game StarCraft II", The AlphaStar team, January 24, 2019. [Online]. Available: https://www.deepmind.com/blog/alphastar-mastering-the-real-time-strategy-game-starcraft-ii[AccessedMay31,2022]

[2] R. orthonormal, "AlphaStar: Impressive for RL progress, not for AGI progress", LESSWRONG, November 1, 2019. [Online].
Available: https://www.lesswrong.com/posts/SvhzEQkwFGNTy6CsN/alphastar-impressive-for-rl-progress-not-for-agi-progress [Accessed May 31, 2022]

[3] Anonymous, "Scouting", Liquipedia StarCraft II, 2018. [Online]. Available: https://liquipedia.net/starcraft2/Scouting#References [Accessed May 31, 2022]

[4] Anonymous, "Help:Reading Build Orders", Liquipedia StarCraft II, 2018. [Online]. Available: https://liquipedia.net/starcraft2/Help:Reading_Build_Orders [Accessed May 31, 2022]

[5] L. Jin, P. N. Nikiforuk and M. M. Gupta, "Decoupled recursive estimation training and trainable degree of feedforward neural networks," [Proceedings 1992] IJCNN International Joint Conference on Neural Networks, 1992, pp. 894-900 vol.1, doi: 10.1109/IJCNN.1992.287073. [Accessed May 29, 2022]

[6] P. Vadapalli, "Introduction to Recursive Neural Network: Concept, Principle & Implementation", *upGrad*, Sept. 11, 2020. [Online]. Available: https://www.upgrad.com/blog/introduction-to-recursive-neural-network/. [Accessed May 30, 2022].

[7] H. Xudong. "XiaoSe", *Liquipedia*. (2018). Accessed June 21, 2022. . Available: https://liquipedia.net/starcraft2/XiaoSe

[8] Pang, Z.-J., Liu, R.-Z., Meng, Z.-Y., Zhang, Y., Yu, Y., & Lu, T. (2019). On Reinforcement Learning for Full-Length Game of StarCraft. Proceedings of the AAAI Conference on Artificial Intelligence, 33(01), 4691-4698. https://doi.org/10.1609/aaai.v33i01.33014691