CSCI599 Spring 2020

Final Report

iTower

Abstract

The tower defense game is a well-known strategy game. Many machine learning algorithms like deep reinforcement learning have been developed to learn how to play a tower defense game. However, according to the research papers that we found through Google, not many studies have tried to use supervised learning like CNN to train the agent. In our study, we built two CNN models, one is a classification model while the other is a regression model. We would discuss all the details that we make to the original game as well as the whole workflow. The results show that the CNN regression model can produce a better result than the CNN classification model. This study gives us a new direction in the area of game development.

Game Introduction

Game Concept

This 2D tower defense game is developed in Unity (C#). In this game, Monsters walk from the blue portal on the top left to the red portal on the bottom right. And the player needs to use coins (collected by killing monsters) to purchase different types of towers and place the tower properly to prevent monsters from reaching to the end of the road.[1]

Target Audience

All age groups

Genres

iTower is a single-player, 2D, tower defense game.

Objective

To reach the next level, the player current life must remain greater than 0 after the last monster of this level is exterminated or arrives at its destination.

Game Details

More game details and modifications could be found in Methodology - Game Design.

Project Objective

The goal for this semester is to create an agent that could learn to play the game wisely. Based on the aspects of current gold (numbers), current monsters (numbers and types), towers (costs and types), the agent should conduct the following possible behaviors: placing towers at proper time and location, selecting the proper types of towers, upgrading properties of the towers.

Project Guideline

- Several necessary modifications of the original game, such as monster routes, tower placement. Details could be found in Methodology - Game Design.
- The setup of a reliable network for the game and the python agent transmitting data between each other via TCP. Details could be found in Methodology - Communication Design.
- An efficient method to collect the raw data from the gameplay. Details could be found in Methodology - Raw Data Collection.
- An efficient method to pre-process and label the collected data. Details could be found in Methodology - Data Pre-processing and Labeling.
- The proper application of TensorFlow 2.0 (Keras) and Convolutional Neural Network (CNN) for the game project. Details could be found in Methodology - Machine Learning Design.

Prior Research

Over the years, machine learning has been widely applied to game development. Many research papers have been published in this area, and a lot of new algorithms have been developed to enhance the performance of the game agent. This revolution gives us a new idea and direction on how to improve the mechanism to make the game more smart and unique. Many traditional machine learning methods, as well as deep learning, are applied in game development. One famous example is AlphaGo, which developed by DeepMind, owned by Google, uses CNN and search trees to play Go. Two other early examples are Waston and Deep Blue, which were both developed by IBM.

When we focus on game development, reinforcement learning, including Q-learning and deep Q-learning, is more widely used than other machine learning methods. For instance, in a game similar to Super Mario, one general idea of playing this game is capturing several screenshots, and then building a deep Q learning model. This algorithm is efficient in this game, and the pattern is not difficult to detect. Many websites offer users the environment to test their reinforcement learning model. Some companies open the API for machine learning that allows the users to call it directly. This policy helps developers save a lot of time because there is no need for the developers to look inside the details of the game. Unity has released a machine learning model called NavMash and a Python-based library, which allows the developer to train their game under the Unity environment.

Unity has provided many demos with existing algorithms for machine learning, where we can download directly from Github. These games already include some built-in API that can be called directly with Python. However, our group wants to do something so that we choose a raw game (the game without any machine learning model inside). After doing research, most games we found are using reinforcement learning as the basic algorithm. Many reinforcement learning algorithms are applied in strategy games, especially real-time strategy games, like tower defense games, Civilization IV.[2] Some tower defense games are 3D, and the mechanism inside is quite complicated. However, the mobile version of the tower defense can also be used to train agents and build models, which is easier to understand compared with the former one.[3]

Although it is possible to simplify these strategy games, a lot of studies have proved the issues that we must face about. For instance, In tower defense games and other real-time strategy games, the Agent needs to make online decisions according to the environment change. The Agent inside the game must make successive attempts without enough information.[4] Another challenge for Agent to play real-time strategy games is the Agent has to learn in an unsupervised manner in a complex, non-adaptive environment.[5] Train Agents in real-time strategy games are time-consuming and labor-intensive.[6] One solution is to use a high-level strategy, which is chosen from different game situations and a fixed opponent strategy.[7]

Not only reinforcement learning, but a large number of other algorithms can also be used on tower defense games and other strategy games. Tower defense games offer many choices for researchers. Some papers focus on increasing the number of monsters killed[8], others might be the method to classify the components inside the game, like using image recognition to detect the monsters and towers.[9] One study applied a genetic algorithm with two different neural networks: Feed-forward and Elman Recurrent.[10] Hybrid case-based reasoning (CBR) is also a good choice, where spatial environment information is abstracted into a number of influence maps. [11][12]

Reinforcement learning might be a good choice right now, but might not be the best. CNN is also used in game development. CNN regularly works on image classification. When the images are resized to small resolution, the accuracy of the model might reach 0.7. Now we have a general idea: CNN might work if the game contains a map. However, we do not find many studies on CNN applications on tower defense games. Tower defense game contains a map, which allows the developer to convert to the matrix (similar to an image, but the number of layers might be different). We found a game on Github. We guess CNN shall work because the map of the game is smaller, and the mechanism is more straightforward compared with other tower defense games. We try to only focus on the smaller, easy 2D games.

In this study, we mainly focus on using the CNN model to train our tower defense game. To prove our idea, we use both the CNN classification model and the neural network regression model.

Tools

- Keras
- TensorFlow 2
- Anaconda
- Python 3.7
- Unity
- C#
- JSON

Methodology

Game Design

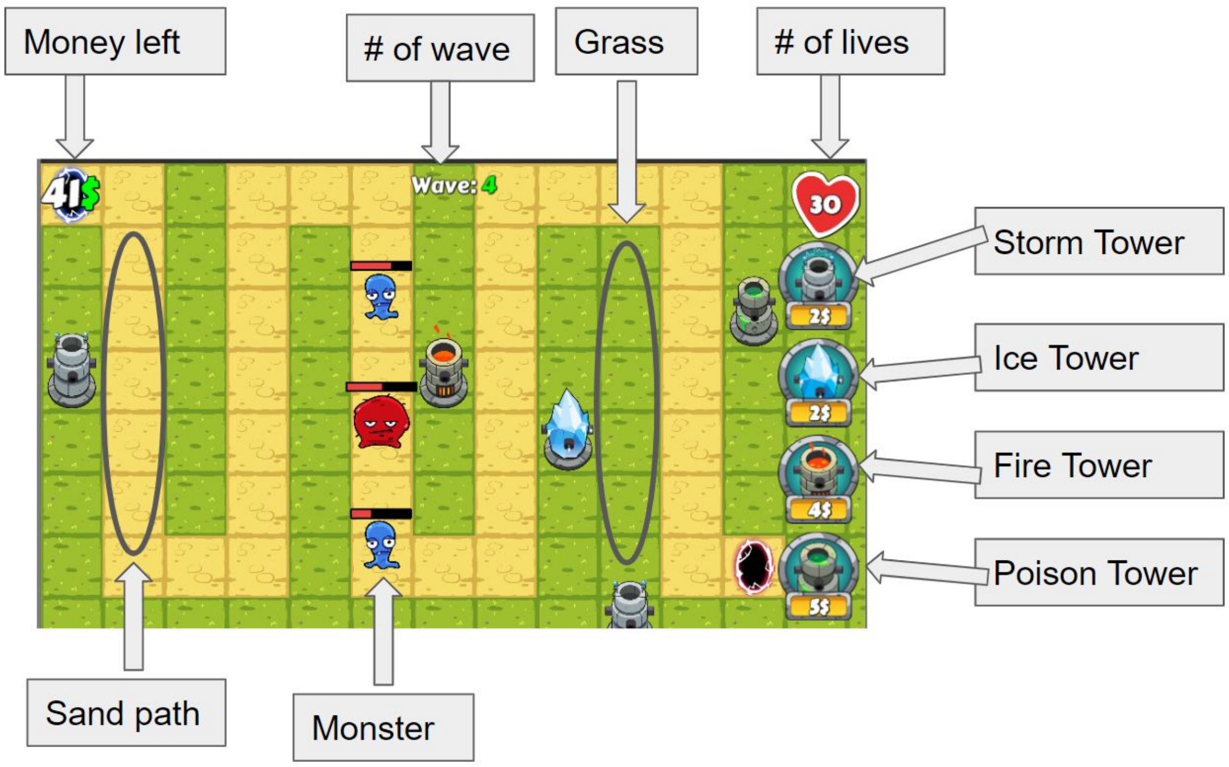

Figure 1. The user interface of the tower defense game

The game we work on is a tower defense game that can be found on Github. It is built with Unity and C#. The game contains a map whose size is 12 (in X-axis) and 8 (in Y-axis). The overall number of tiles on the map is 96, including green ones (grassland) and yellow ones (sand path).

Four types of towers are offered in the game: storm tower (labeled as 1), ice tower (labeled as 2), fire tower (labeled as 3), and poison tower (labeled as 4). The price for each tower is presented below that tower. One difference of this game is it shall not be considered as a continuous game since it could pause automatically after each “wave”. The monsters are generated as a queue, but they would not come out together at once. Instead, a fixed number of monsters would come out each time, which equals the number of waves. For instance, two monsters would come out in wave 2.

The initial money and lives, and the map are defined in the game. But we modified the game so that it can listen to the Agent, and fetch the parameters from the JSON data stream. For instance, the session called generate data, each game would start with 500 coins and 50 life points. Once life points reach zero, or there is no place for towers, then the game ends. The game would generate a spawn portal and a target portal. Monsters would be spawned in the spawn portal and move toward the target portal on the designated path. The monster number would be incremented as the wavenumber increases.

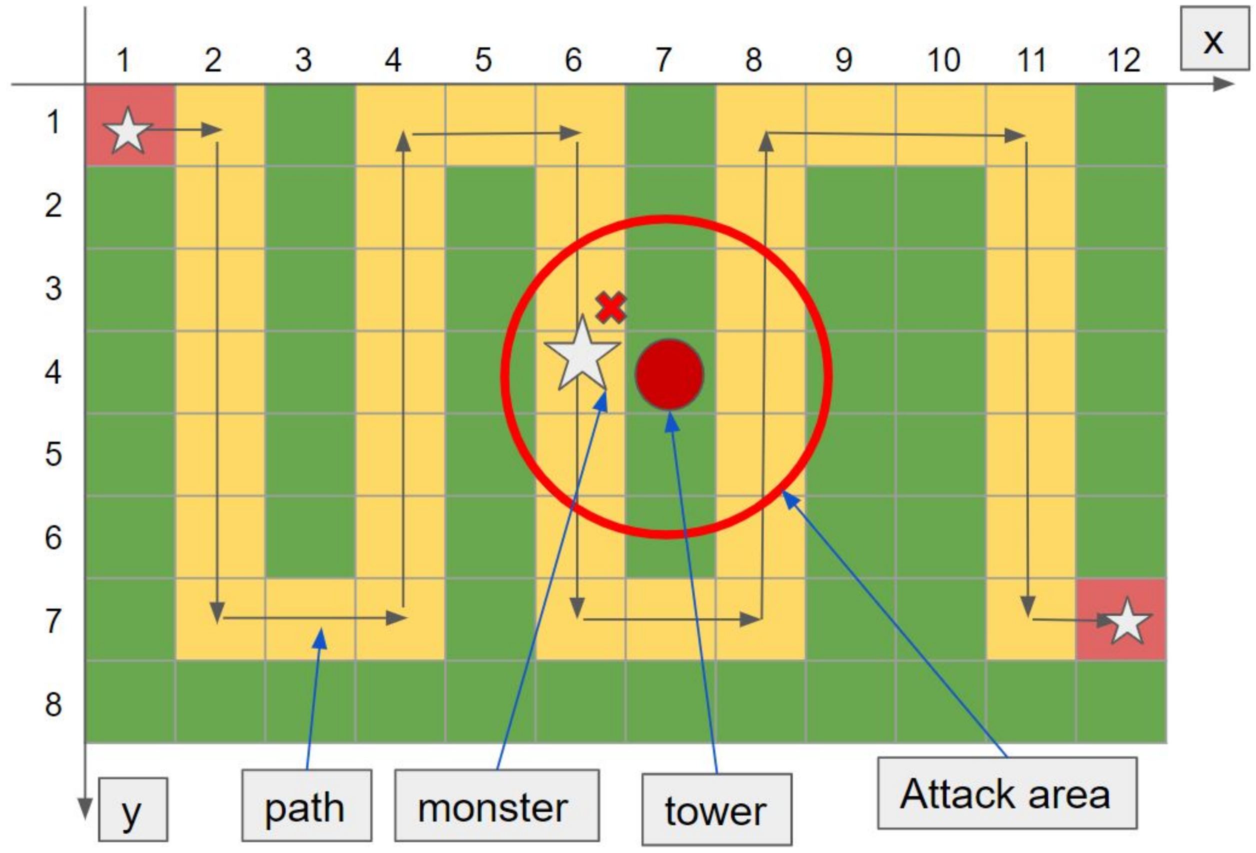

Figure 2. The illustration of the game. The grey arrows represent the route of each monster, and the monsters can only walk on the sand path. The smaller circle represents a tower, and the larger circle represents its attack range. Note 1) the monster might not be killed if it was attacked only once; 2) we have modified the game to extend the attack range. In the latter study, we use coordinates to represent the position of each tower and each monster.

Refer to figure 2, here are several things we have changed in this game. 1)The path of each monster in the original game is based on the A-star algorithm, but we modified the game to make them move within a fixed path. A monster would not be killed unless it is attacked by one or more towers at the same time. 2) there is some limitation on the number of towers that can be placed in each wave. We have modified the game so that the range of the number of towers placed in each wave is 2 to 4, but users can change these parameters directly in the Python code. 3) In the original game, the function that places the tower is triggered by the mouse click. But this time, the function of placing towers is redesigned as an API that can be called by the C# program. This modification has also been applied to selling towers and upgrading towers. 4) After doing research, we find data send and data receive in C# cannot be invoked in the main thread. So we create a new thread that can handle the data transmission independently. 5) The Python agent has a thread pool so that the python program can play multiple games together in a multithreaded environment without human intervention.

- *Life and Money Modification:*

Before we modify the game, the **Game_Manager.cs** can load the **GameData.txt** file in the Resource folder. The life and currency value to be initialized for the game are written in this file. The format is “life, currency”. The Game_Manager would use these values to start the game. Now, we let the agent send initial money and lives as parameters to the game.

- *Tower Instantiation Modification:*

Modify the code for **Game_Manager.cs / TileScript.cs**. Towers are instantiated in the function `placeTower` of the Game_Manager. By modifying the placeTower function, the python program is able to instantiate new towers at the beginning of the game without complete previous “click button - track coordinate - left-click map - instantiate” instantiation cycle. Instead, the python program can simply generate the coordinates of the tower and directly call placeTower to place a new tower in the target grid. We call the new function PlaceTowerAPI.The interval of the number of towers that can be placed in each wave is 2-4.

- *Map Modification:*

Assets/Scripts/Managers/LevelManager.cs is responsible for initializing maps for each game trial. Map config is currently stored in Resource/LevelData.txt. Blue and Red portals are spawned by SpawnPortals() in LevelManagers.cs. The map could be changed by editing those files. In the previous iteration, the map is fixed, and the learning process is based on this map. Now, the game map is sent directly from Agent to the game, which means we can modify the map without changing any code in the game.

- *Monster Life and Speed:*

Monster speed can be modified based on the level to increase the difficulty of our game. The monster life changes may need to be designed in a non-linear pattern. As initial 3 HP is impossible for monsters to reach the red portal, and the terminal firepower of towers may be reached quickly by a linear increase of monster HP.

- *Monster Amount:*

Monster amount increment speed is doubled to 2 per wave to prevent too many perfect game results, which are not useful for learning progress. The wavenumber does not go as 1,2,3,4.., but go like 2, 4, 6, 8…

- *Error handling:*

In the original game, the master might be stuck due to some internal errors or bugs caused by the author on the Github. We set a max threshold time for each wave to allow the game to restart automatically if it detects some errors.

- *Dependency Removal:*

This part is important when we use the CNN regression model. Normally, the next tower placement would depend on the tower placement if we only call place tower API. That means if we play multiple games together, the tower placement in one wave cannot be converted to any other wave if they are not in the same game. Now we have three APIs, place tower API, sell tower API, and upgrade tower API. To reduce unnecessary operations, in each wave we must follow the order: place tower->sell tower->upgrade tower.

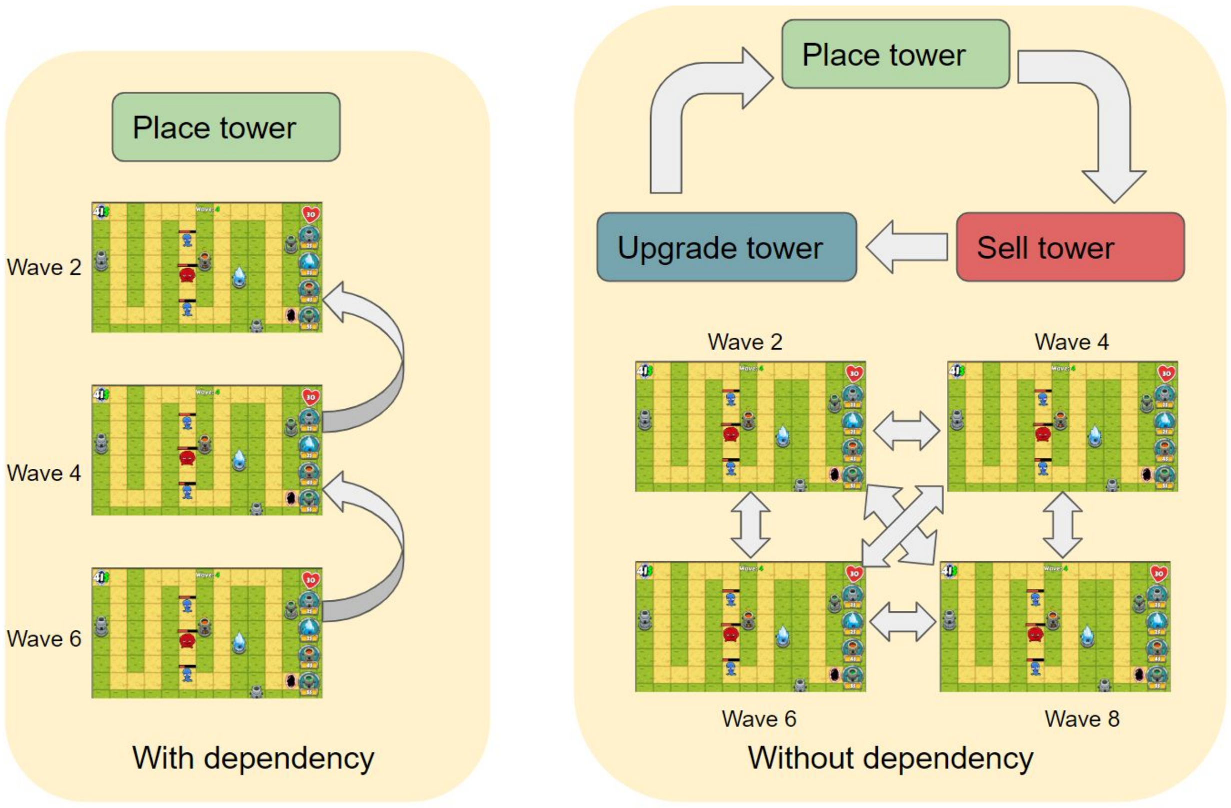

Figure 3. The basic workflow of with/without dependency

When we use the CNN classification model, it does not matter which mechanisms we use. But if we switch to the CNN regression model, we must consider the dependency.

- *Reward function:*

In the original game, there are two parameters that we can use to verify if the tower placement is good or not: money left and lives left. But the two parameters might conflict with each other because they are complementary. If we kill more monsters, it is likely that less money would be left, and vice versa. It is not guaranteed that we can kill more monsters and earn more money at the same time. Thus, we redesign the reward function. The equation includes two parts. The former part is called min remained path length. Suppose there are two monsters: blue monster and red monster. The blue monster is killed after going through 19 cubes. The total length of the path is 40 cubes. Then the blue monster has 21 cubes left. However, suppose the red monster has been killed after going through 35 cubes, then the remaining length is only 15 cubes. According to the reward function, we would only choose the min one, 15 cubes. On the other hand, because both two monsters are killed before they reach the end, the left lives remain unchanged, still 50 lives. Now the final reward value would be 15+50 = 60.

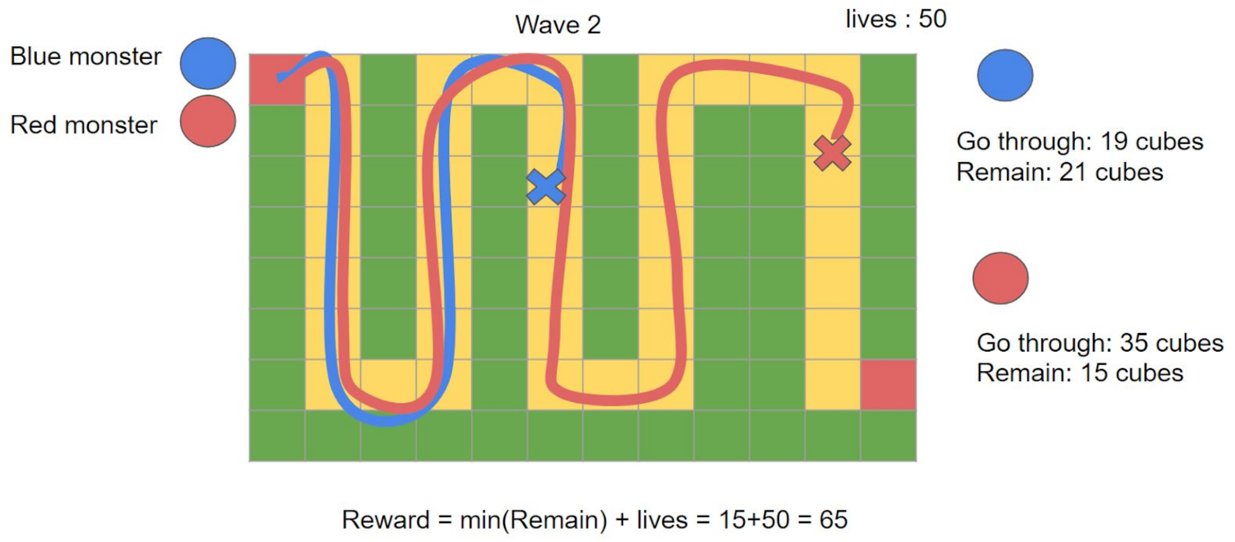

Figure 4. The basic mechanism to calculate the reward. The reward function can be divided into two parts: min(remain length) and lives

- *Game termination:*

In the previous session, the game would terminate if there is no money or lives left, and it might cause a problem. The user might still lose the game even though he killed a lot of monsters but no money left. To solve this problem, we increase the initial money from 50 to 500, and the initial lives from 10 to 50. The agent can place as many towers as it can, but the total number of towers cannot exceed the max threshold (when there is no grassland left). After doing that, the max number of waves that the agent can still survive is wave 20 or even 22, while the early session is only 10. In a word, there are two ways to terminate the game: no lives left, no place for a new tower.

- *Game Flow Summary:*

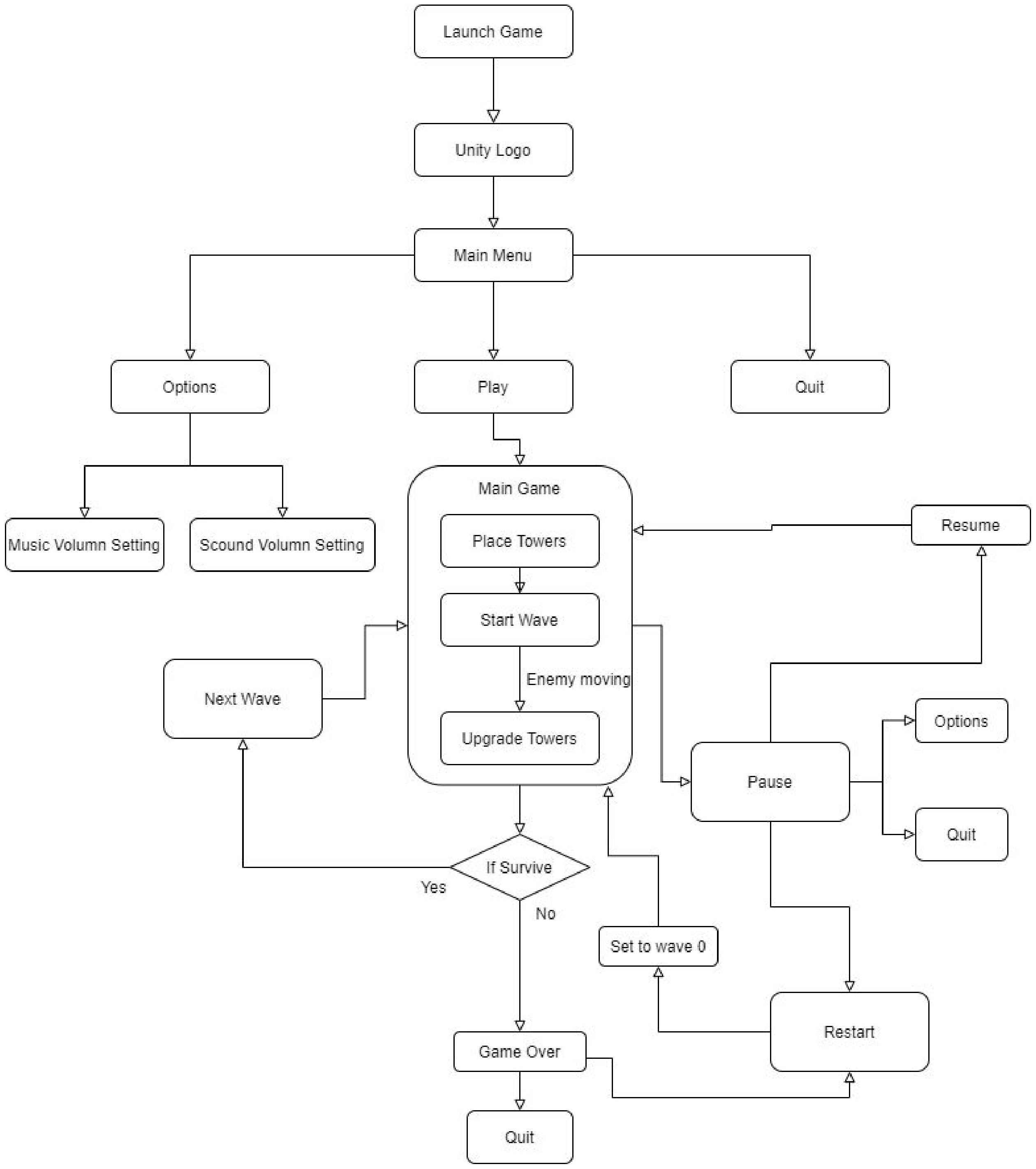

Figure 5. The overall workflow of the game section. This flow chart describes the basic mechanism of the game.

Agent Design

The Agent we designed is written in Python, a better language in machine learning. The environment we use is Ubuntu 18.04, with Anaconda 3.7. The Unity company has released the Linux version of Unity Hub, which makes it possible to switch the target of the Unity from Windows/Mac to Ubuntu without modified code. We also installed Rider from JetBrains, a cross-platform C# IDE.

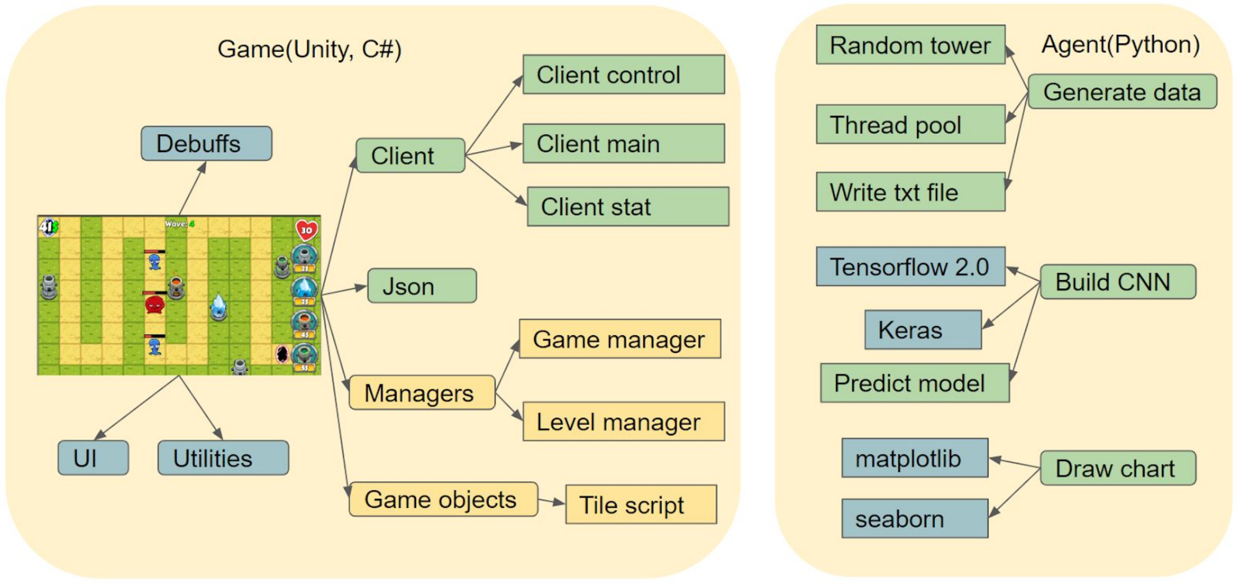

Figure 6. The basic structure of the Agent (right part). The Agent has three main functions: generating data, building CNN, and drawing charts. The green parts represent the code written by iTower team members, the blue parts represent the code from external libraries, and the yellow parts represent the code modified by iTower team members to enable specific functions related to the project need. Note: the original game we found on Github does not have any API to communicate with external programs via TCP or other protocols. We must modify the structure of the game to enable its data exchange function (we would discuss the details in the Communication Design part).

Refer to figure 6, there are three main functions in the Agent, and each function has several sub-functions. 1) The Agent can generate data. 1.1)The game can not create data automatically until an agent communicates with it. Because no data has been recorded before, the Agent would play the game randomly (generate the coordinates and type of tower randomly). 1.2) The Agent would write text files after it receives the data (for details, please refer to the Communication Design). The agent would collect and store the data into text files (for the details of the structure of the text file, please refer to Pre-processing and Labeling part). 1.3) The Agent has a threading pool to enable multiple threading. The data collection part is time-consuming, and the average time to play one round is 1 minute. The threading pool allows the Agent to play multiple games at the same time. The maximum clients (game) that the Agent (server) can connect to is 25, according to our test. The number might be larger if the computer has more CPU and memory. 2) We use CNN as our machine learning method. 2.1) We use Keras API in TensorFlow 2.0 (for details, please refer to the Machine Learning Design). 3) the Agent would analyze data and draw charts. 3.1) The python graph library we use is matplotlib.

- *Agent Flow Summary:*

We designed two models for the agent, one is a CNN classification model, the other is a neural network regression model. Note: the regression model could be considered as part of the CNN classification model. We would mainly focus on the neural network regression model because the performance of this model is much better. We would also compare the difference between these two algorithms.

As we mentioned in Agent design, the whole machine learning algorithm is put inside the Python agent, including data generation, model training, model verification, and data analysis. The Unity game itself does not include any machine learning models itself. Instead, we build an API that allows the game to communicate with the agent, like receiving and sending data in JSON format.

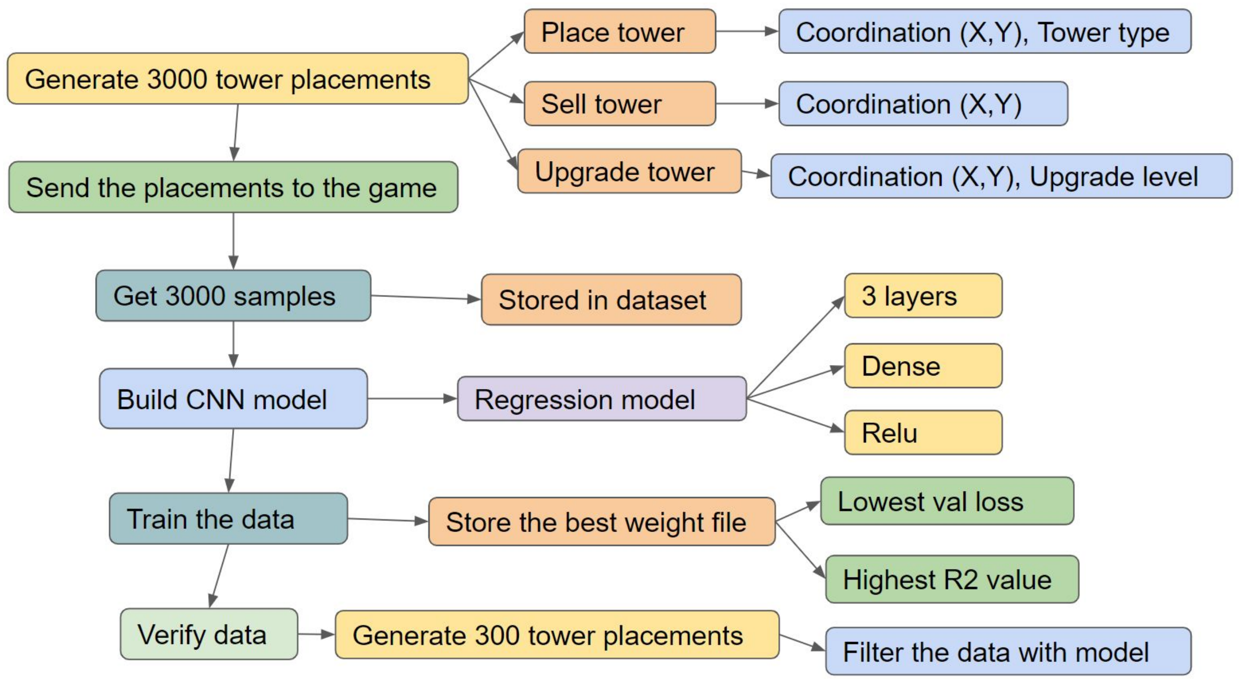
Generate 3000 tower placements
Place tower
Coordination (X,Y), Tower type
Sell tower
Coordination (X,Y)
Upgrade tower
Coordination (X,Y), Upgrade level
Send the placements to the game
Get 3000 samples
Stored in dataset
Build CNN model
Regression model
3 layers
Dense
Relu
Train the data
Store the best weight file
Lowest val loss
Highest R2 value
Verify data
Generate 300 tower placements
Filter the data with model

Figure 7. The basic workflow of the Agent (NN regression mode). The whole process can be divided into four main steps: generate data, train model, verify model, and analyze data.

This workflow describes the first three sessions. In step one (generate data), the random generator inside the agent generates 3000 tower placements and sends them to the game. Each sample includes tasks: place tower, sell towers and upgrade towers. Place tower: choose any one of the four types of towers, put the tower onto the game map, represent the tower with an index number. According to the coordinate system we build in the game design session, the data structure to represent the placement is clearly described in figure 7. If there is no error, the reward and wave numbers would be stored together with the placement map as JSON format files. Thus, about 3000 JSON files would be stored in disk for step two. In each wave of the game, one coordinate on the map cannot be used twice, which means all the random coordinates are unique.

In step two (train data), the regression model is built with Tensorflow and Keras. One difference between this regression model and the CNN classification model is the depth. When we load the 3000 JSON files back from the disk, the game map is stored as a 3D NumPy array. Before importing into the networks, the high dimension array needs to be flattened as a one dimension array and plus the wavenumber. The depth of the networks becomes small but its width increases a lot (Please go to the machine learning session to see the details). Another difference is the data set. We divide the whole 3000 data into the train set and the verify set. After 30 epochs training, we get 30 weight files (Normally, the model would only store the last weight only. However, we add a callback function so that the weight file after each epoch is stored as well) and select the one with highest R2 value (R2 value is used in regression problems).

In step three (verify data), the neural network would load the weight file (the best one) we created in step two. The Agent will generate another 1000 samples for model verification. In this step, the networks work as a filter to only maintain the top N samples with the highest reward.

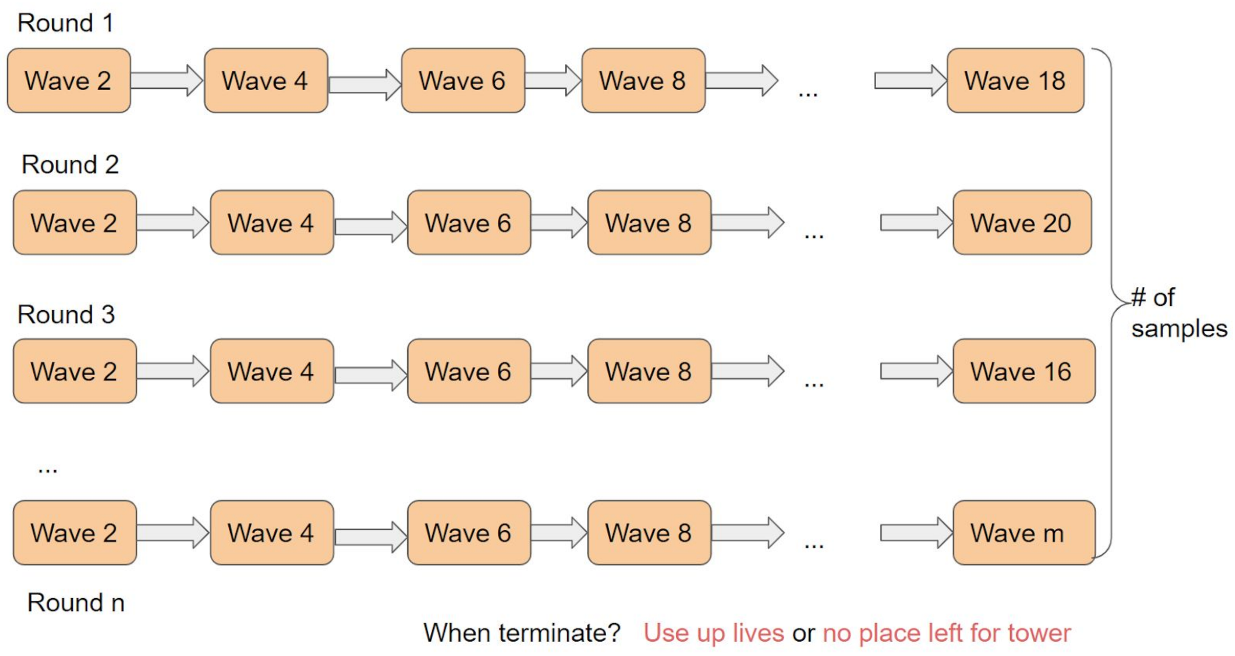

Figure 8. The basic relationship between wavenumber and the number of samples. In this case, the value of n equals to 3000, but the values of m are varied. The minimum value of m could be 10 while its maximum value could reach 20.

If we consider the whole data collection as a matrix, then the index of a sample would be a row and the wavenumber would be a column. Due to the variety of tower placements, the number of waves that Agent can survive is also different. There are two mechanisms to terminate the game: use up lives, no place left for a new tower.

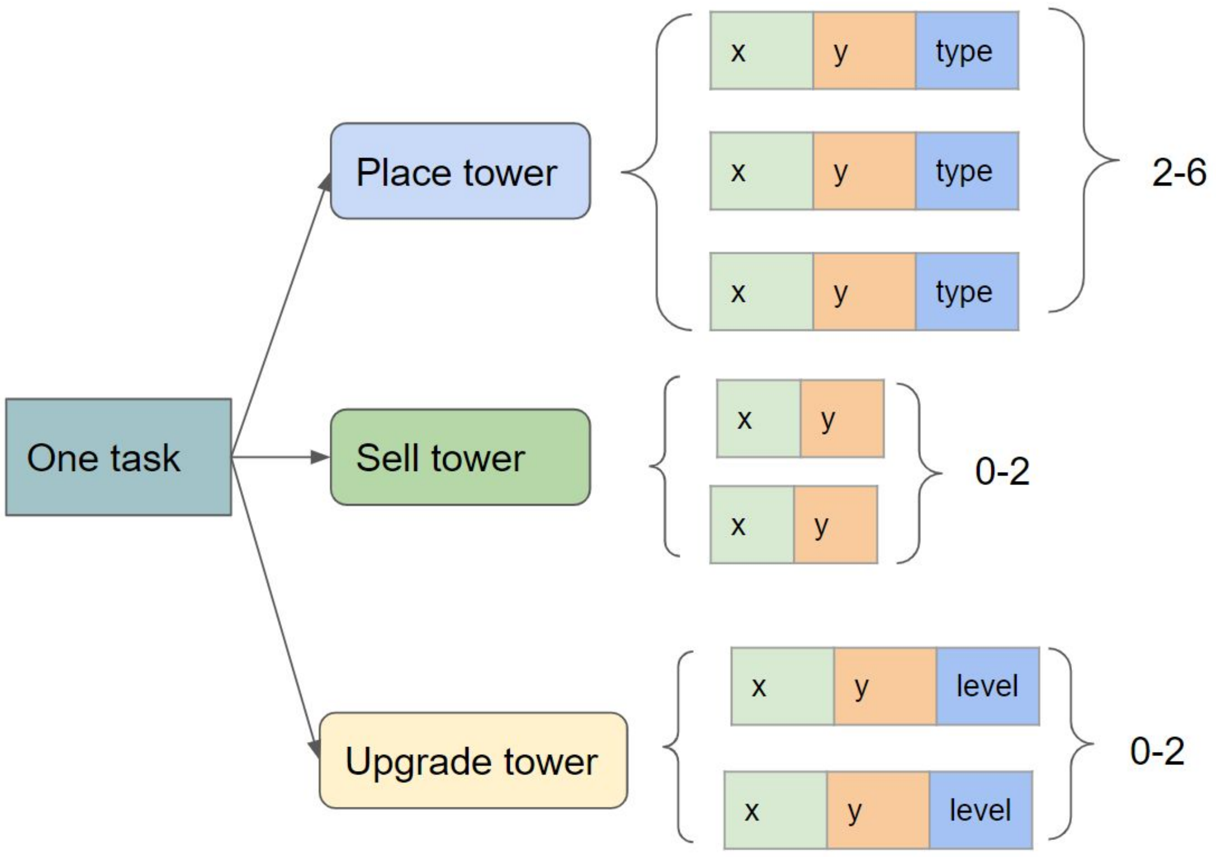

Figure 9. The basic structure of a JSON file that sent from the Agent to the game

As we mentioned above, each sample that is randomly generated by the Agent contains three tasks: place towers, sell towers, and upgrade towers. To narrow down the range of the random generator inside Agent, we set some internals to restrict the randomness. For instance, the minimum number of placing towers is two while the maximum number is six. The fields in each object are also different. When upgrading towers, we do not need to consider the type of tower that is already placed, but the upgrade level. To make it convenient, we only allow each tower to be upgraded at most twice.

Communication Design

The communication design session is a key point in our project. The original game we found on Github can only be controlled by a mouse. It cannot communicate or exchange data with any external programs. In this session, we discuss the details about the communication API we add to the game and how it works.

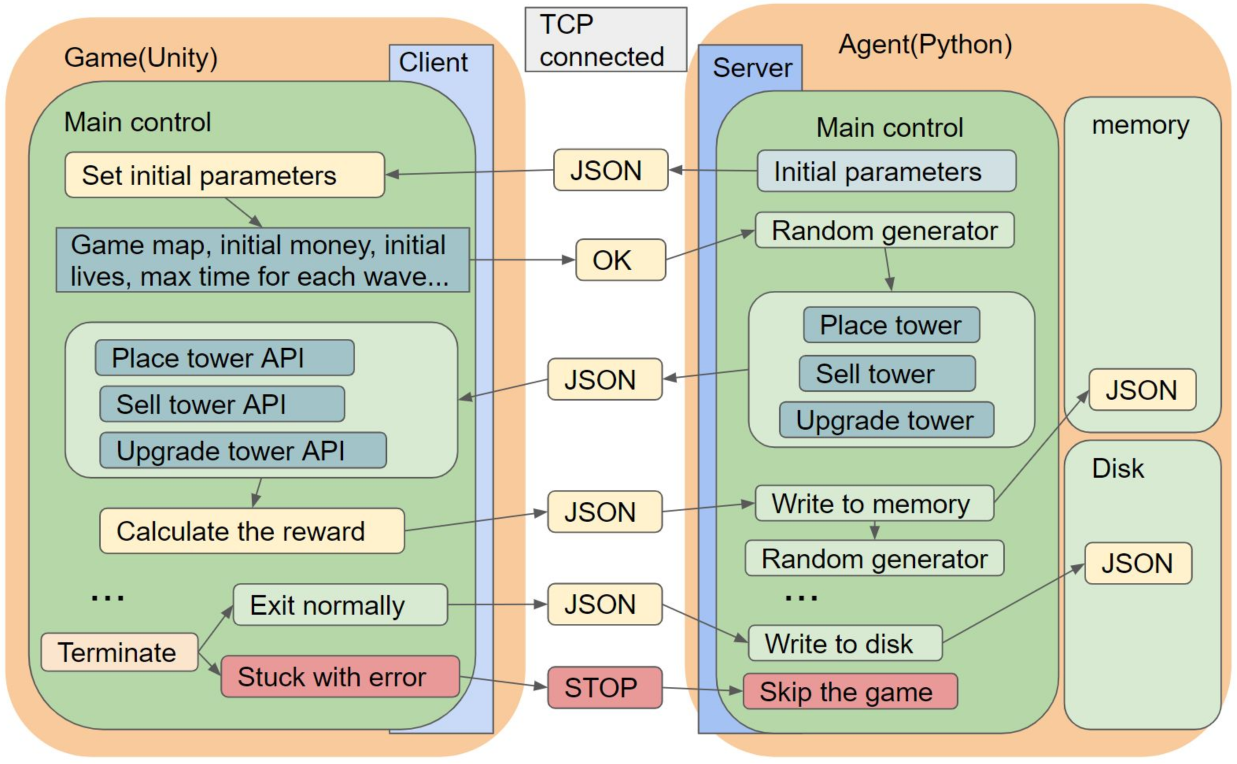

Figure 10. The mechanism of data exchange between Game (Unity) and Agent (Python) via TCP.

We modified the game to enable the TCP client function. The Agent works as a server-side. When both games and Agents listen to the same IP address and the port number on one computer, the TCP connection would be established. Data exchange between the client and server is JSON format encoded in UTF-8. The first data sent from the Agent to the game is the initial parameters. The initial parameters include all the necessary parameters as a JSON format that the game needed. In the original game, the game map is built inside the game, but now we allow the Agent to generate a game map randomly and send it to the game before playing it. The initial money and lives are also determined by the Agent rather than the game itself. The reason to do that is to improve the flexibility of our project. When the game finishes one operation, it would send “OK” back to Agent, waiting for the next operation. The random generator is the key function inside the Agent that used to generate new tower placements that we mentioned above. When the client of the game receives these JSON files, it would call three APIs in the main thread: Place Tower API, Sell Tower API, and Upgrade Tower API. At first, we try to simulate the mouse click, but later on, it was proved to be a not effective idea. These three APIs are quite complex because we need to rewrite the basic mechanism inside the game. After calling all three APIs, the value of the reward is calculated and send back to the Agent. Due to the internal errors that we found inside the game, the game might be stuck due to some unknown bugs. Thus, the temporary data would not be written into the disk until the game exit normally. This mechanism works like a loop, it would terminate when one game reaches the end. In the final step, the Agent collects all previous data, writes a report (in JSON) as a text file.

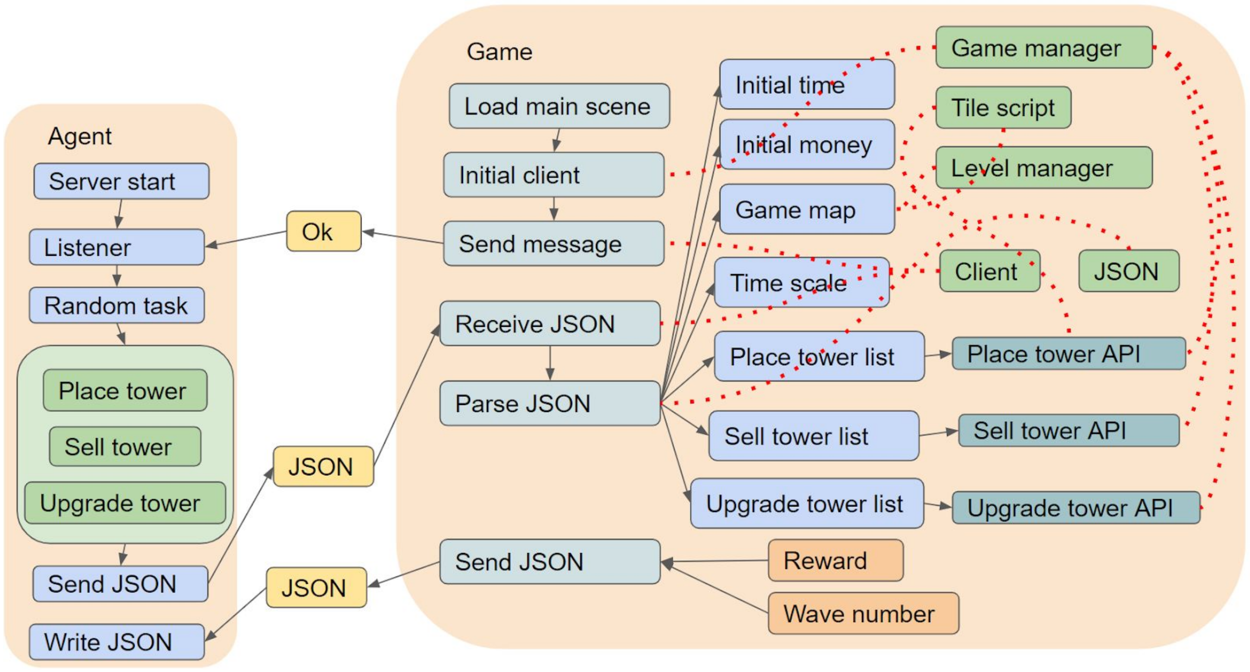

Figure 11. The detailed process inside the game when placing a tower.

In our project, the Agent works as a server which needs to be started before the game. After loading the main scene, the socket would be created between these two programs to exchange datastream. Figure 11 describes the top five scripts in the Unity game, including the game manager, the tile script, the level manager, the JSON, and the client. The game manager is the main thread. It covers the three APIs that we mentioned above (placing towers, selling towers, upgrading towers) and the socket connection. The tile script and level manager are used to create the game map for the game according to the data received from the Agent. The client must be built on a child thread (the whole game would be bothered if we put the client on the main thread). The client has four main functions, receiving data, parsing data, converting data, and sending data. The parsing data function parses the JSON string to C# objects. The converting data function is opposite to the parsing data function, it would convert the C# objects to JSON strings.

The data collection process is time-consuming. Enabling multiple threading can speed up the process. The maximum number of games that one agent can play at the same time depends on the computer. If the computer's CPU has 14 cores, 25 games might still be fine. For a normal computer or virtual machine, 4-8 games might be the maximum.

Raw Data Collection

As mentioned in the Communication Design, an automatic agent has been developed to place towers, play the game, and generate the training data for the machine learning part. The tower placement is completely random, but we designed strict criteria to select the applicable training data.

As mentioned above, we build two models to train our data. The first one is the CNN classification model. The wavenumber that we allow the game to play is fixed; usually, we set it as 5. The latter one is the neural network regression model. The wavenumber in this model can vary.

Convolutional Neural Network Classification Model:

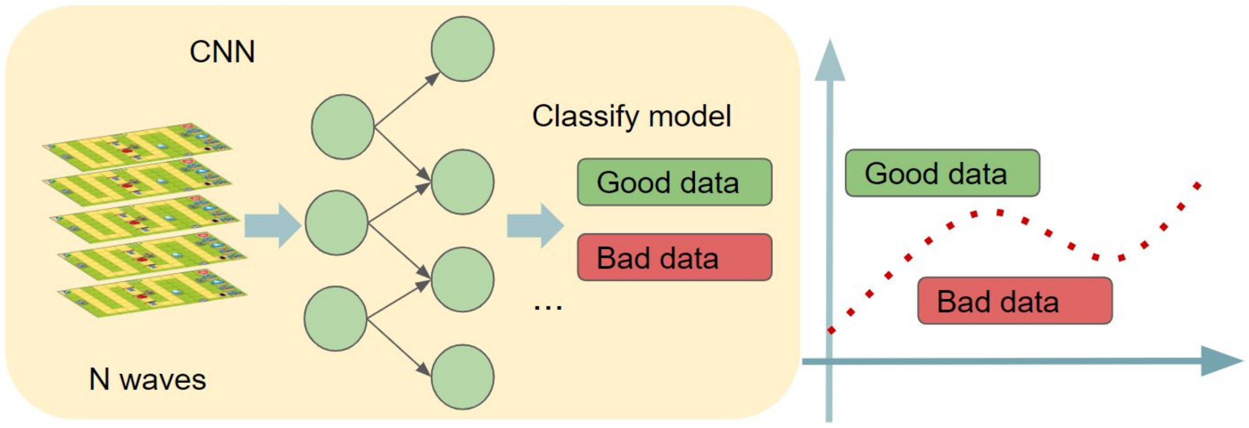

Figure 12. The CNN classification model

To ensure there is sufficient data for the training part, we generated over 8,000 groups of raw data by using the automatic Agent to play the game within five waves. The data would have the following information:

1. The total number of monsters killed: stands for the total monsters who have been killed during the 5 waves.
2. Game map: is a 3D matrix with game_map_size* wave_number (currently is 72*72*5) stores the current map of the tower defense game. It has information about tower type and tower location.
3. Money left: the total money left after playing the game after 5 waves. (Kill monsters would earn money and buying towers would spend money)

Deep Neural Network Regression Model Based on Reward:

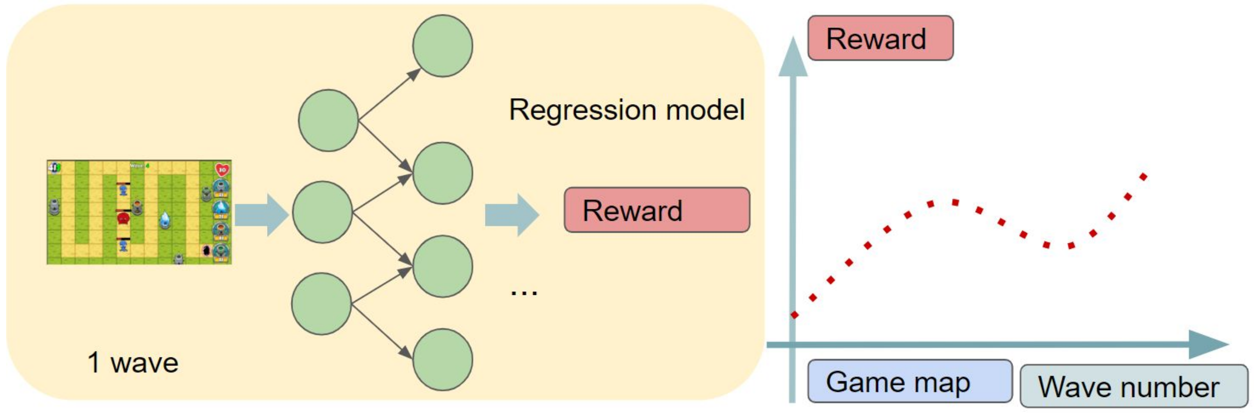

Figure 13. The neural network regression model

To ensure the data is sufficient for the neural network regression training, we collected about 3,000 datasets by using the automatic Agent. The agent keeps playing the game until the game is over. The data would have the following information:

1. The tower placement information including the tower type as well as tower position. (8*12*4 3D matrix)
2. Wavenumber to store the state of the game.
3. Total life remaining.

Data Pre-processing and Labeling

Convolutional Neural Network Classification Model:

To ensure the data is good for the training, a pre-processing step is implemented. After careful data selection by every team member, 1,100 groups of data are selected based on the following criteria:

1. It can successfully play 5 waves.
2. It can kill a relatively high number of monsters (above average).
3. It can earn relatively more money (above average).

To label the data, a cutoff binary labeling method is incorporated. After reviewing all the data after pre-processing, the data performs above the average (higher than the average monsters killed & higher than the average money left) are labeled as 1, and the other data are labeled as 0.

label	**Explanation**
1	The dataset with a higher number than the average monsters killed and more money than the average money left.
0	The rest are labeled by 0.

Table 1. The mechanism of labeling good/bad data in the CNN classification model

For the training data, approximately 80% of the 1,100 data would be used. And the rest 20% of the dataset would be used for validation and tuning parameters for our model.

Deep Neural Network Regression Model Based on Reward:

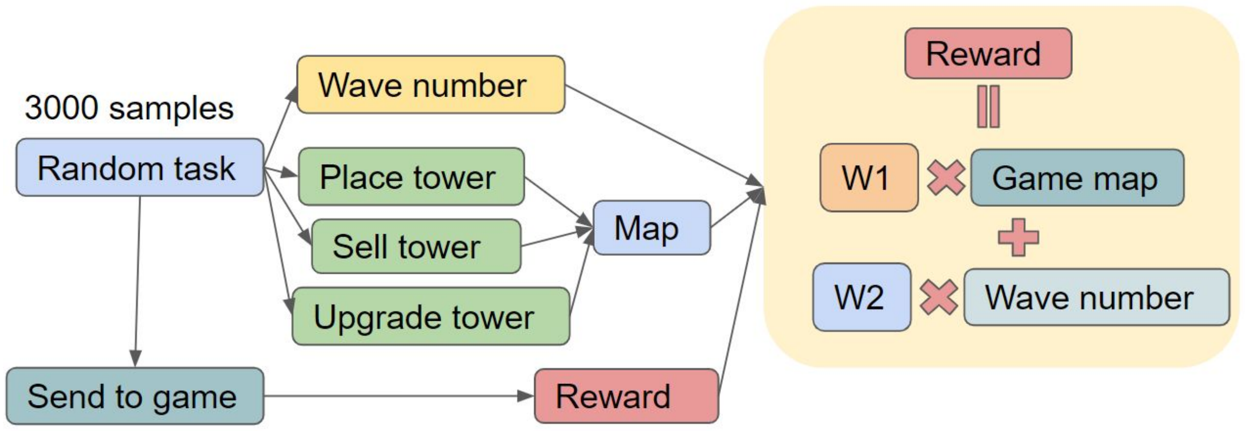

Figure 14. The reward function (game map, wavenumber)

After exploring various methods of reward function designs, we found the combination of total life remaining and the farthest position that a monster can reach is the best indicator of the game performance. The game has a higher reward value indicates the game has been playing well.

$$\begin{aligned} Reward &= Total\ life\ remaining + Monsters'\ shortest\ remaining\ distance\ to\ the\ end \\ &= F(game\ map,\ wave\ number) \end{aligned}$$

We collect the randomly generated tower placement data combined with wavenumber as well as total life remaining for each wave and monster' shortest remaining distance to the end. The following steps are implemented for the data preprocessing:

1. Eliminate the incorrect data due to the game bug.
2. Normalized the tower placement data using min-max scaler normalization
3. Computed the reward for each wave

For the training of NN, we have about 3,000 data. 90% of the data has been used and 10% was utilized for the model validation.

Machine Learning Design

The machine learning framework is TensorFlow 2.0 [13] (Keras [14]) for our project. Currently, we incorporate 2 models to train the model to learn the best strategy to play the Tower Defense game we created. We initially chose CNN because the game map for the tower defense game is very similar to an image, which is a common input for CNN.

In the first half of the semester, we use CNN as a classifier to classify good and bad strategies. With exploring ways to improve the overall performance, we found the regression model with deep neural networks can provide better outcomes with a refined reward function. The regression model is used to predict the game reward. The two model structures are presented below.

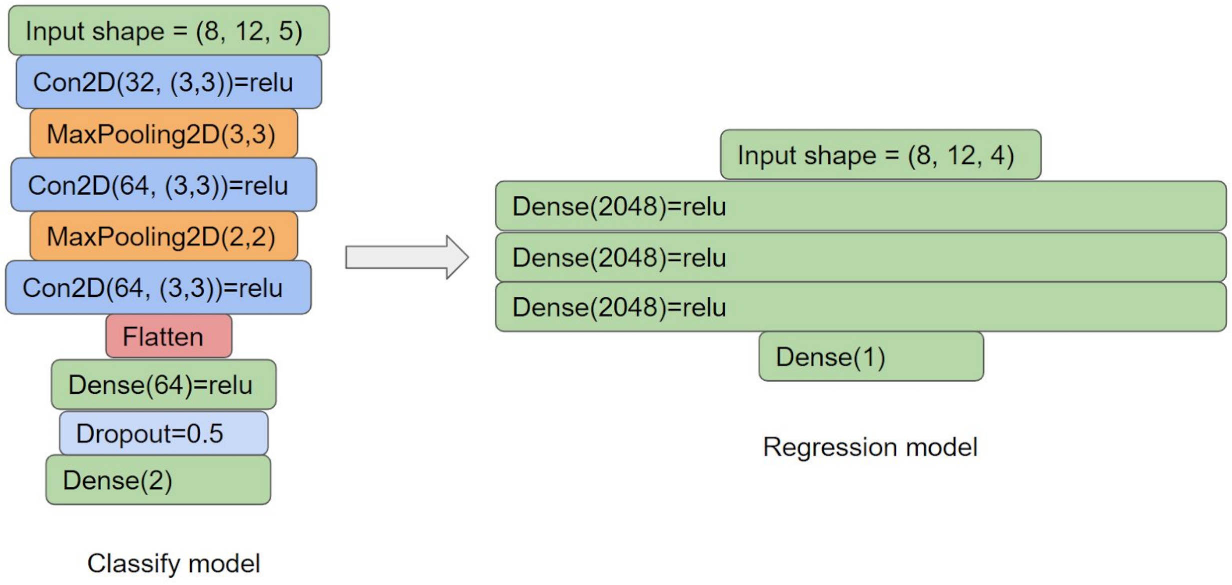

Figure 15. The neural network regression model we use could be considered as part of the CNN classification model.

We choose the **Deep Neural Network Regression Model** as our AI algorithm based on the following criteria:

1. It uses a single metric (reward) comparing the double metric (money & number of monsters killed) in the CNN model. The single metric is more explicit and easier for evaluation.
2. The regression model is an end to end model. The CNN model is more complicated as it needs to recognize the map first by filters. Less bias occurred in the intermediate states for the regression model.
3. Overall, the regression model performs much better.

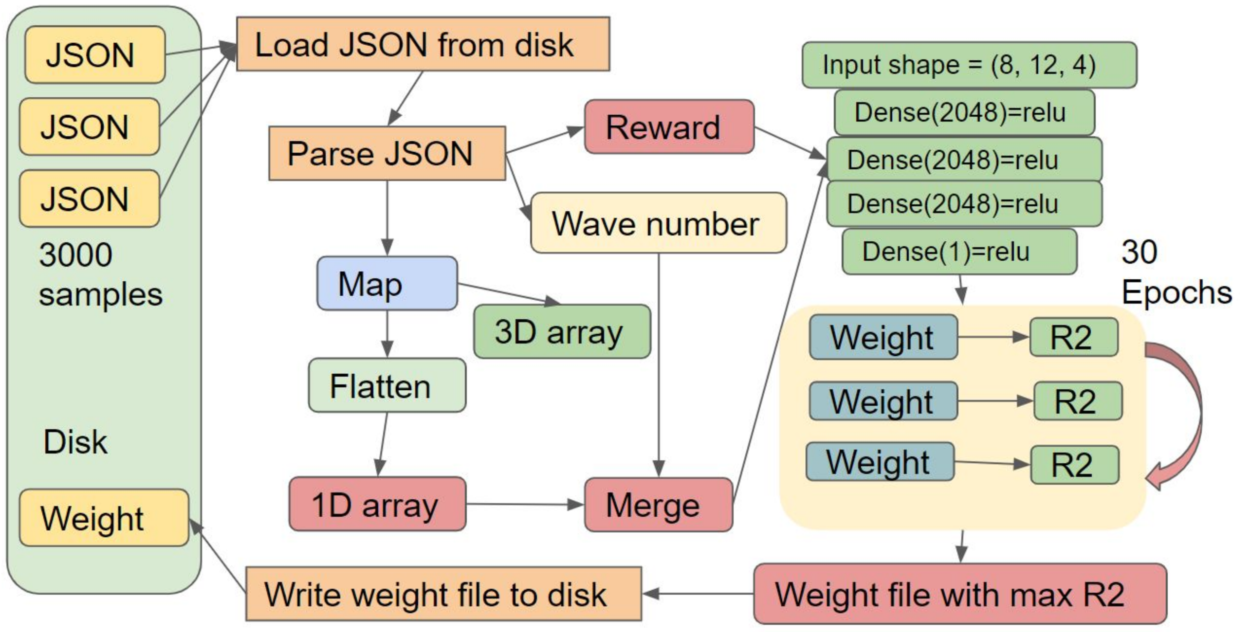

Figure 16. The mechanism of training data from 3000 samples, generating weight files

The Agent first loads all the 3000 samples (JSON files) from the disk. The JSON parser in the Agent can convert the JSON string to Python objects, including the game maps, rewards, and the wavenumber. Because the dimension of the game map and wavenumber is different (game map is 3D array while the wavenumber is an integer), we need to flatten the game map to a 1D array and merge it with the wavenumber. Then the 1D array is the input of the neural network. We set the training epochs to 30. Only the weight file with the highest R2 (min loss value) would be stored.

Hyperparameter Tuning for the Deep Neural Network Regression Model
The model design is shown below. We choose R-squared (Coefficient of determination
) as our metric for the hyperparameter tuning because it is a statistical measure of how close the training data are to the regression result. The higher R-squared achieved by the model indicates the regression model is more close to the actual scenario.

R-Squared value		Neurons Per Layer				
		128	256	512	1024	2048
Network Depth	3	0.75	0.80	0.85	0.85	0.86
	5	0.82	0.82	0.83	0.83	0.84
	10	0.80	0.83	0.82	0.81	0.80

Table 2. Max R2 value of the neural network with different depths and number of neurons

Neurons per Layer	3
Network Depth	5
Activation Function	ReLU
Dropout Rate	0
Optimizer	Adam
Epoch	30

Table 3. Hyperparameter tuning results

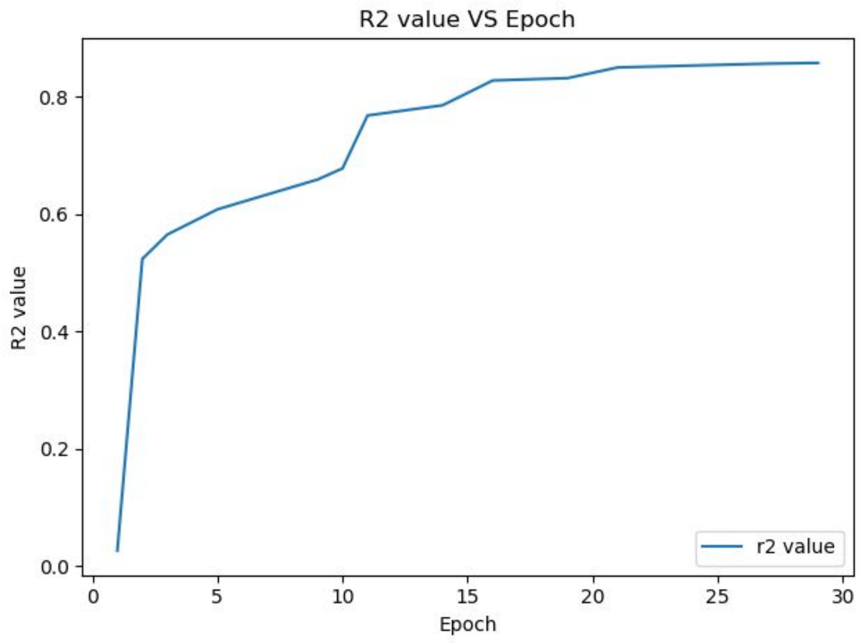

Figure 17. The tendency of R2 value when we set the depth=3, number of neurons per layer=1024

The regression model with three hidden layers and 2048 neurons per layer, as well as 30 epochs, can achieve the highest R-squared value = 0.86. As the value is very close to 1, it means our model prediction result is very close to the actual scenario.

Game Strategy Generation

For each wave, the agent will generate up to 1,000 possible tower placement based on the current money available using a backtracking algorithm.

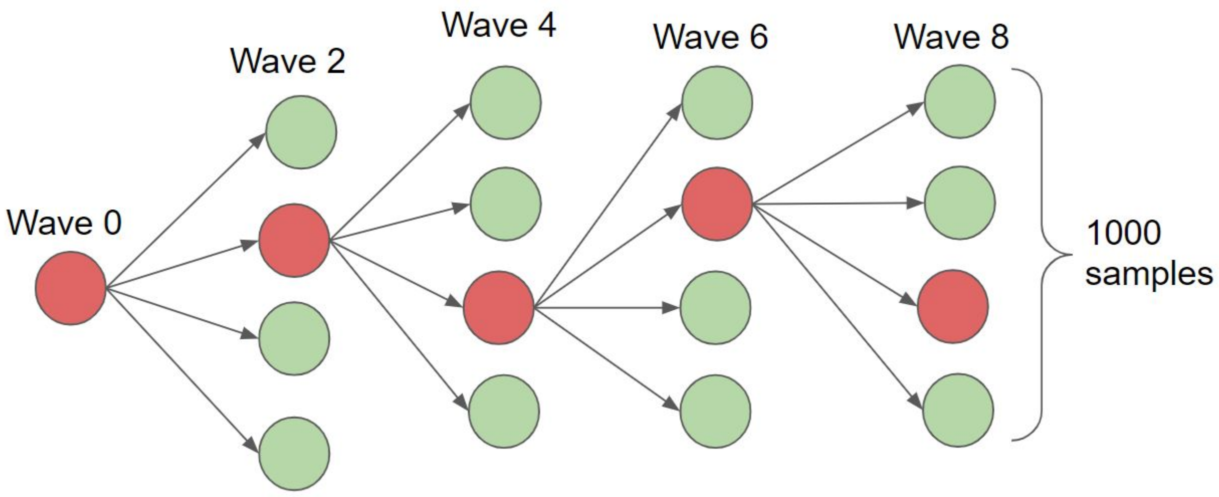

Figure 18. The whole filtering process can be considered as a tree (red means highest reward)

We put the potential tower placement dataset generated above into the regression model, the corresponding reward is predicted by the model. The tower placement with the highest reward will be selected as the move for the next wave.

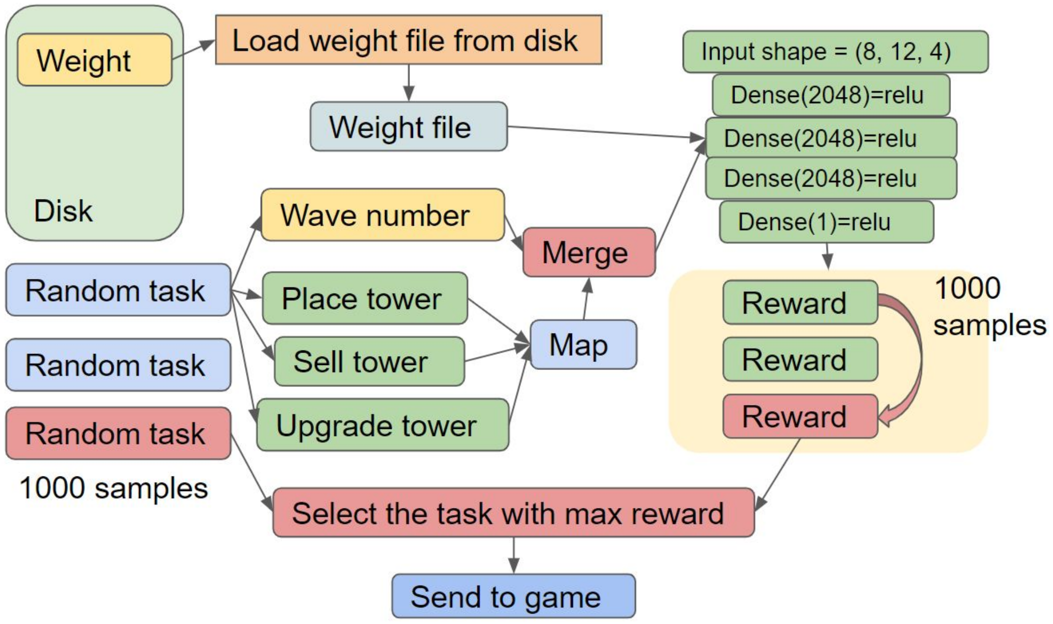

Figure 19. The mechanism of verifying data by generating another 1000 samples each wave

The Agent first loads the weight file that we generated in the previous step. Then we use the same function that we used to randomly generate data to create another 1000 samples. The model we build can predict the reward of each sample and only maintain the best one. That means only one of 1000 samples would be on the left and doing further processing. In each wave, the Agent would generate 1000 samples until the game exit (used up lives or no place left for a new tower). The similarity between figure 18 and figure 19 is the game map needs to be flattened from a 3D array to a 1D array before importing to the neural network.

Performance Evaluation & Conclusion

The regression model can significantly improve the performance comparing the CNN model and other random generated strategies, which can play the game well and survive more waves.

To prove our idea, we set two groups. One is before the training (represented as blue), which means everything is random; the other is after the training (represented as yellow). In the survival rate line chart, the maximum survival wave before training is 20 while it reaches to 22 after training. When we look at wave 20, the value of yellow is much higher than the blue line. However, the yellow line also drops down after wave 20 because we only have the data until wave 20.

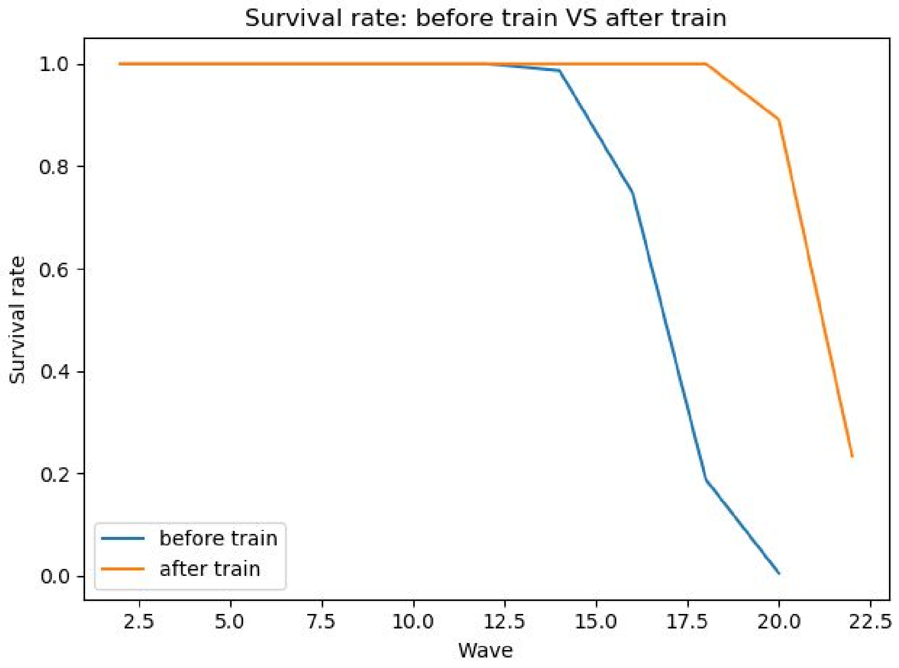

Figure 20. The survival rate before the training and after the training

Based on the reward perspective, the regression model also significantly boosts the reward of every wave. The graph below shows the reward distribution comparison of the model performance before training and after training.

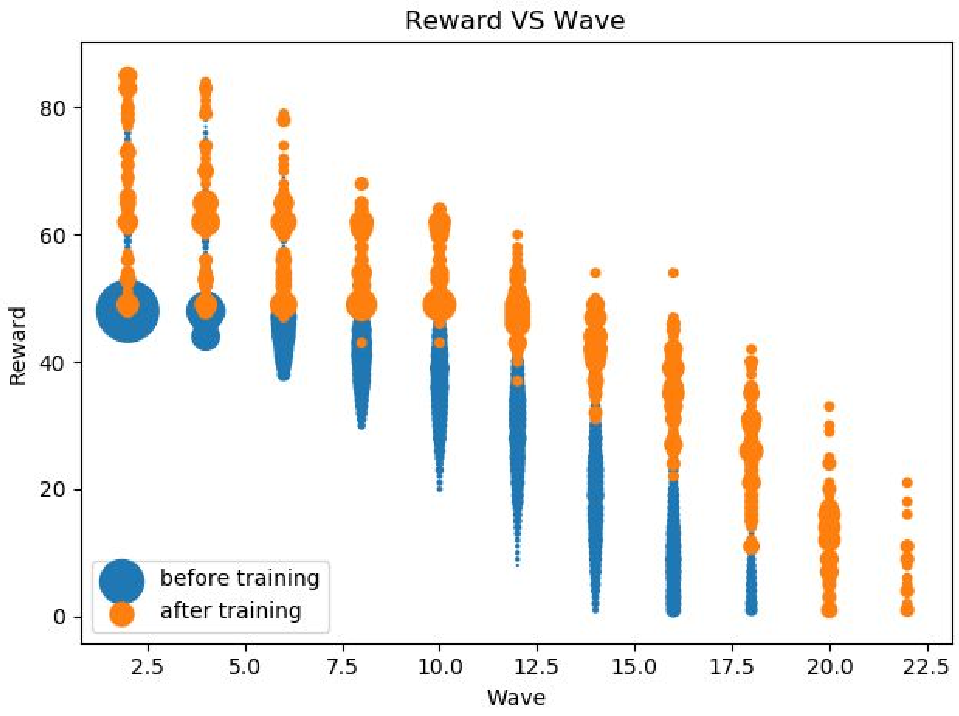

Figure 21. The reward before the training and after the training

Overall, the strategy generated with Deep Neural Network Regression Model can play this tower defense game with more waves than a general human and some machine learning models. The process is challenging but we reached our initial expected goal.

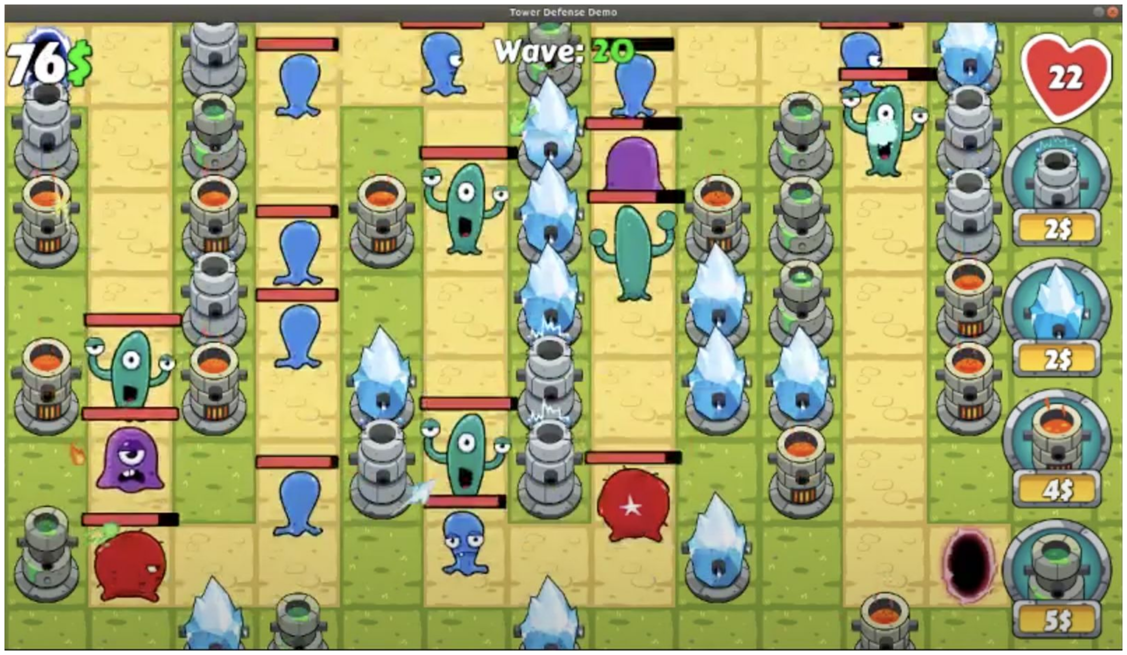

Figure 22. A screenshot of the game after training

Limitations & Future Work

There is a minor bug in the original game we found in GitHub, the bug may cause the game to freeze. But that situation doesn't occur very frequently. As our team is mainly focused on the game redesign and refine the AI algorithms, we haven't solved this bug yet. The bug affected some training data and may influence the trained model. Besides, as all the data is acquired using the same map, we are wondering to extend our model to various game maps in the future.

In the future, our work will be:

- Fix the bug caused the game to freeze.
- Collect more training data after the bug has been fixed.
- Explore to improve the model performance using newly collected bug-free training data.
- Extend our model to more maps.

References

[1]. Jesse Huang, 2D-Tower-Defense https://github.com/JessHua159, 2017
[2]. S. Wender, "Using reinforcement learning for city site selection in the turn-based strategy game Civilization IV," Computational Intelligence and Games, CIG, 2008.
[3]. A. M. H. Wong, "Game layout and artificial intelligence implementation in mobile 3D tower defence game," *International Journal of Security and Networks,* 2015.
[4]. P. A. Rummell, "Adaptive AI to play tower defense game," International Conference on Computer Games (CGAMES), 2011.
[5]. S. Wender, "Applying reinforcement learning to small scale combat in the real-time strategy game StarCraft:Broodwar," Computational Intelligence and Games, CIG, 2012.
[6]. S. Liu, "Automatic generation of tower defense levels using PCG," 14th International Conference on the Foundations of Digital Games, 2019.
[7]. C. Amato, "High-level reinforcement learning in strategy games," in 9th International Conference on Autonomous Agents and Multiagent Systems, 2010.
[8]. P. Massoudi, "Achieving dynamic AI difficulty by using reinforcement learning and fuzzy logic skill metering," International Games Innovation Conference (IGIC), 2013.
[9]. P. Avery, "Computational intelligence and tower defence games," IEEE Congress of Evolutionary Computation (CEC), 2011.
[10]. T. G. Tan, "Automated Evaluation for AI Controllers in Tower Defense Game Using Genetic Algorithm," International Multi-Conference on Artificial Intelligence Technology, 2013.
[11]. S. Wender, "Combining Case-Based Reasoning and Reinforcement Learning for Unit Navigation in Real-Time Strategy Game AI," International Conference on Case-Based Reasoning, 2014.

[12]. B. Auslander, "Recognizing the Enemy: Combining Reinforcement Learning with Strategy Selection Using Case-Based Reasoning," European Conference on Case-Based Reasoning, 2008.
[13]. TensorFlow https://www.tensorflow.org/
[14]. Keras https://keras.io/, https://github.com/keras-team/keras/

Team

Name	Position
Wen Ni	Project Manager, Agent Designer
Licheng Jiang	Game Designer
Hao Jin	Game Designer
Nuo Xu	Agent Designer
Enda Zhang	Agent Designer
Junjin Wen	Agent Designer