# Project Report

**TAM, Ka Ho (1155175983) and LAM, Hei Yui (1155176788)**

Group 5

## Overview

Reinforcement learning has always been a hot subject in the field of artificial intelligence. To explore the capabilities of reinforcement learning, our group developed a shooter game *SnowFight* and trained the agent to play it. We will be covering the details of the game and the training in this report.

## Game Design

### *Goals*

*SnowFight* is a shooter game in which the player controls the main character to survive in the zombie apokalypse. There are two main goals in the game: survive as long as possible while killing as much zombies as possible.


Figure 1: Gameplay

### *Actions*

The player could move forward and backwards, turn left and right by pressing the arrow keys. The player can also throw snowballs by pressing the space bar. For convenient training using deep Q-network, only one action can be taken per step. On the other hand, as human players would often press multiple keys at once, in this case a random action is selected from the pressed keys[1].

### *Behavior of Characters*

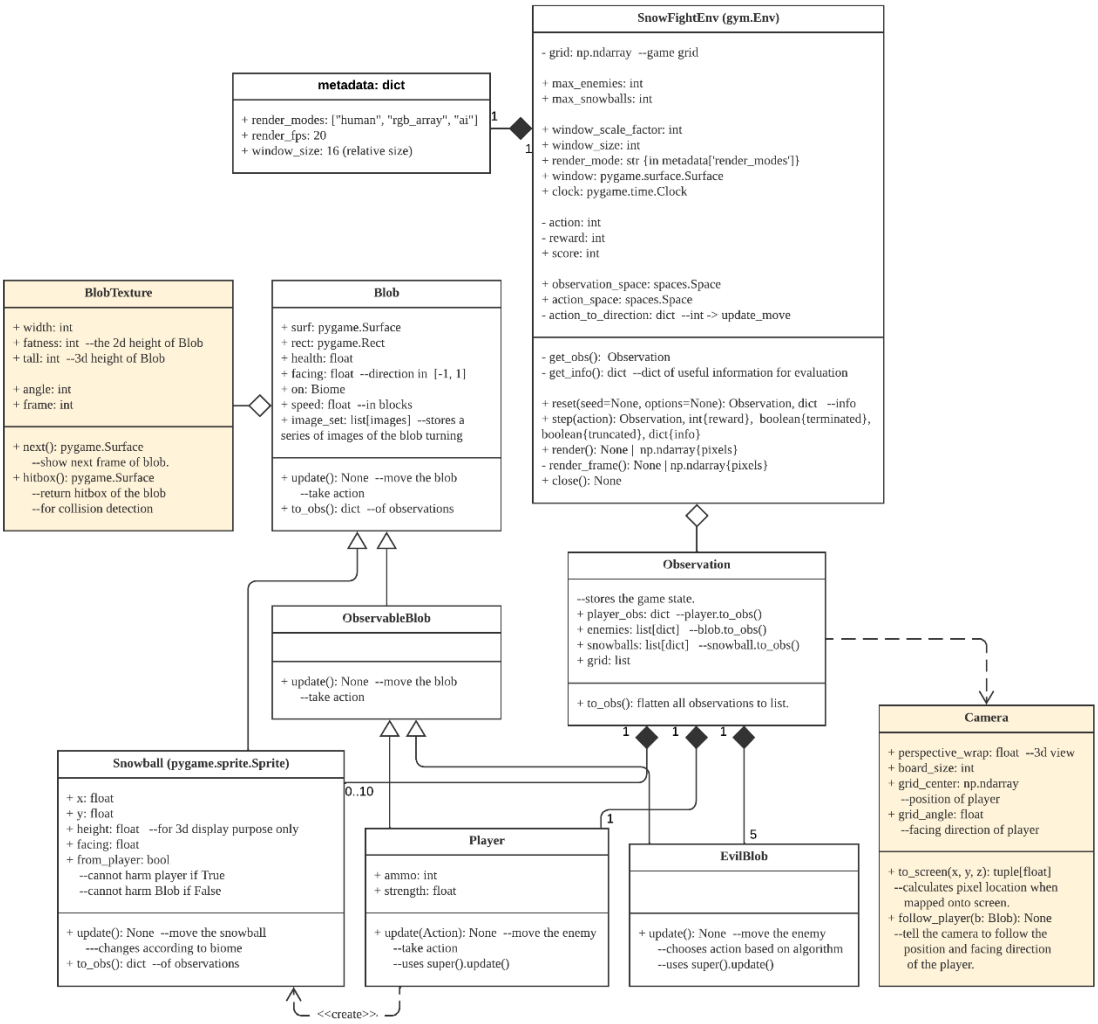
In the game, the zombies are enemies which would slowly turn and move towards the player. They have the same action-control mechanism as the player, which means that they could not turn and move simultaneously. Players can harm the zombies by throwing snowballs at them, reducing their health points by 1. Three hits are required to kill a zombie, after which a zombie would be randomly respawned in another location at least 10 units away from the player. As the game progresses and as the player gets higher and higher scores, the zombies would speed up gradually. The player is dead when a zombie collides with the player.

[1]Please note that due to hardware limitations, simultaneously pressing 3 or more keys might NOT work. Some key combinations like ‘up’+‘left’+‘space’ might be common, but some keyboards cannot recognize the pressed keys correctly. It is advised to not press 3 or more keys at once while playing.

On the other hand, snowballs thrown by the player have parabolic trajectories and would disappear after hitting the ground. This means that the attack range of the player is limited to around 12 units. Meanwhile, the number of snowballs on the screen is limited to below 10 for easier training of the model.

### *Class Designs*

The detailed class designs are given in the following class UML diagram:




The classes related to the main game are separated into three categories: the gym environment (*SnowFightEnv*), the logic environment and the display environment (shaded in orange).

The logic environment is mainly contained in the python files *blobs.py*, *observable.py* and *observation.py*, which are responsible for most of the updating of the game state. On the other hand, the display environment is mainly located in the python file *camera.py*, and is responsible for the display of the game sprites, utilizing the resources in the *res* directory, which is positioned directly below the working directory *snowfight*.

## Training of the Agent

### *Game state and Actions*

Similar to how the player has 6 possible actions, the agents also have an action space of 6 discrete actions, namely *left turn* (0), *right turn* (1), *move forward* (2), *move backwards* (3), *throw snowball* (4) and *idle* (5). Only one of them can be chosen at each timestep.

The game state, on the other hand, is more complex, as every entity in the game environment is passed into a single observation state. For simplicity and code-reuse, every entity (which must extend the *Blob* class) has a dictionary of states, which are flattened into a tuple once the *to_obs()* function is called. The raw game state from the *SnowFight* environment is a space containing a dictionary of such tuples, making up a total of 88 paremeters (given that there are 5 enemies and maximum of 10 snowballs):

```
'player':    (<x>, <y>, <facing direction>, <moving direction>,
               <health points>, <speed>, <ammo>, <strength>),
'enemies':(
                    ( <x>, <y>, <facing direction>, <moving direction>,
                      <health points>, <shooting>, <size>, <type>),
                    # ... for every enemy there is a sub-tuple
               )
'snowballs':(
                    (<x>, <y>, <facing direction>, <owner>),
                    # ... for every snowball there is a sub-tuple
               )
```

When the number of enemies and snowballs are less than their allowed maximum count, empty default values are placed in order for the dictionary to remain in the same size.

Training with such a large input would lead to slow convergence, hence the raw observation is further reduced. The custom observation wrapper class *CompactifyMore* is utilized to greatly reduce the number of observations so as to increase the efficiency of the training. The returned game state is a flattened numpy array, which contains the following 19 observations before flattening:

```
'player':     (<x>, <y>, <facing direction>, <ammo>),
'enemies':(
                    (<distance from player>, <angle from player>, <relative facing direction>),
                    # ... for every enemy there is a sub-tuple i.e. there are 5 * 3 = 15 observations.
          )
```

Since the agent could perform rotation by pressing the left and right arrow keys, polar coordinates is used for training. Angles and distances are measured relative to the forward direction and position of the agent.

***Rewards***

In order to motivate the agent towards our goals of killing more zombies and surviving for longer time, the agent is rewarded 1 point everytime his snowballs hit a zombie, and 1 point is deducted every time he get caught by the zombies.

While testing with the training of agents, it is observed that the need of long-term dependency causes the model to require a much longer training time (around 20,000 ~ 50,000 epochs) to converge. The previous models is likely to get stuck in an idle state, obtaining an average score of -3.0, which is the lowest score possible in the game. Hence, immediate reward of 1 point is also given when the agent shoots in the direction of any enemy.

***Models Used***

The agent is trained using Deep Q-Learning implemented using Tensorflow. The Deep Q-Network (DQN) model used is a simple 3-layer feedforward neural network, while the model parameters are updated using Adam optimizer and a loss function of Mean-Square Error (MSE), with a target q-value of $y^{(i)} = r^{(i)} + \gamma \max_{a'} Q\left(s'^{(i)}, a'\right)$, giving a loss function of:

$$\mathcal{L}_\theta\left(s^{(i)}, y^{(i)}\right) = \left(y^{(i)} - \hat{y}^{(i)}\right)^2 = \left(r^{(i)} + \gamma \max_{a'} Q\left(s'^{(i)}, a'\right) - Q\left(s^{(i)}, a^{(i)}\right)\right)^2$$

In order to increase the efficiency of the training, the model is trained with a batch of 512 experiences at once, using the experience replay method. In *snowfight_qtrain.py*, the agent is first allowed to interact with the environment in the *PlayModel* function with epsilon-decay strategy, and it starts training a number of episodes once enough experiences are collected.

The experiences are stored and handled in the python file *qlibrary.py*, in the format of $\left(s^{(i)}, a^{(i)}, r^{(i)}, e^{(i)}, s'^{(i)}\right)$, which corresponds to the current game state, action taken, reward obtained, whether the game has terminated, and the new game state after the action is taken.

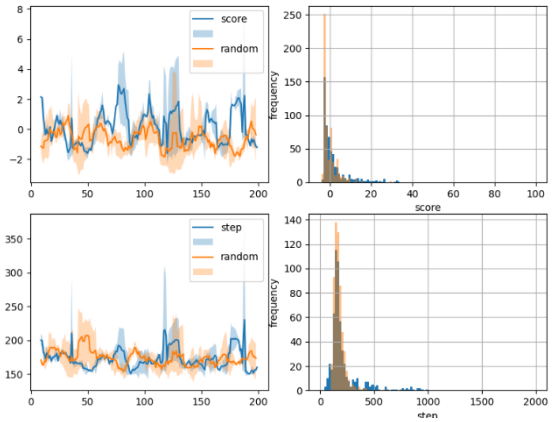
Threading is also used to maximize the efficiency of utilizing the CPU resources. The number of threads in parallel can be set by the user. It is observed that threading has no significant impact on the performance of training, while it has significantly increased the speed of training.

During the training, the models that achieved best scores will be saved to the file *model_best.h5* and are also used to feed more useful experiences to the latest models when they perform poorly.

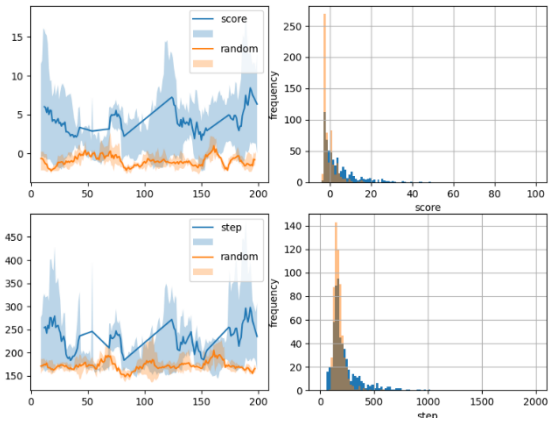
***Hyperparameter Tuning***

To obtain a model that converges quickly, several values of the decay factor (gamma) have been tested. 3 trainings are done for each setting, and below is the graph of the performances of the best models in each training attempt after testing for 200 epochs[2].

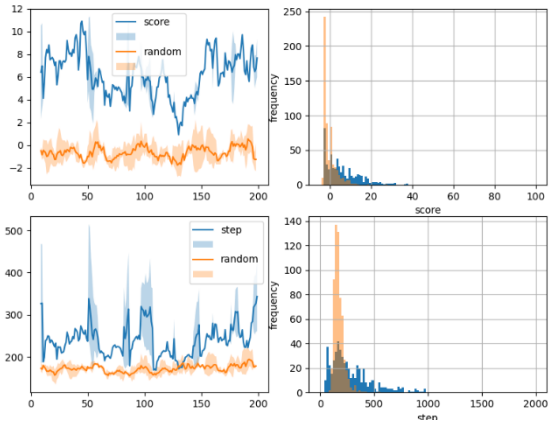
Gamma = 0.99995 (v2)[3]



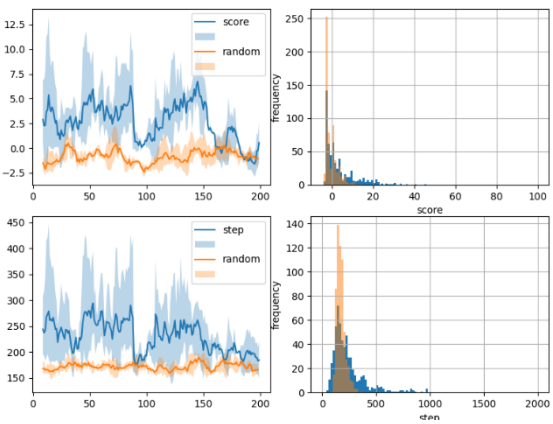

Gamma = 0.9995 (v3)




Gamma = 0.995 (v4)




Gamma = 0.95 (v5)




From the testing, it is observed that the optimal value of gamma is 0.995. While slightly lower values gives acceptable results, when the value of gamma is further increased, the models started to have a high variance in performance and even failed to converge under 1000 epochs of training.

---

[2] Since the plotted graphs are too chaotic, they overlaps each other when plotted on the same graph. Hence, the results are separated into distinct graphs. Hidden layer sizes and learning rates are also analyzed, however, writing them into the report will make this report too lengthy.

[3] The bracketed words denotes the version of the training, and the training results and models can be found in the *data/* folder.

***Results***

While the saved best models from the training are supposed to give the best scores, they might not be using the best strategy. Using the optimal gamma value of 0.995 and learning rate of 0.0001 with no decay, the best scores are generally achieved at around 250 ~ 500 episodes of training. The variance of the number of episodes required to attain an average score of above 10.0 is around 100 episodes:

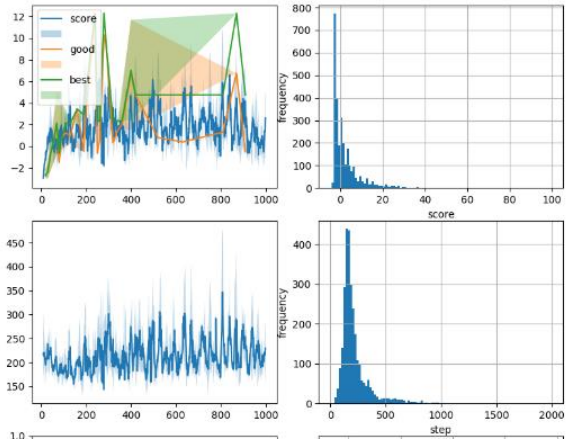



Figure 2: Note that the first few peaks is around the 300th epoch.

Below are the sample runs from the “best models” :


Figure 2 qt_v4/qt_v4_3/model_best.h5

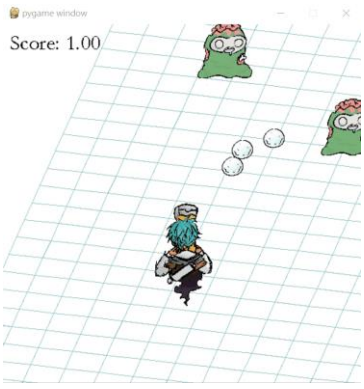



Figure 3 qt_v4/qt_v4_3/model_best.h5

From above runs, we can see that the models learned several strategies like moving to the corner (thus reducing the chance of being chased by zombies from behind) and turning while shooting, which are strategies human players would often use.

## Executing the Game

### *Installing the Package*

Before executing the game environment, it is crucial to note that the game should be opened with *snowfight* as the root folder, such that the resource files can be accessed correctly[4]. The *gym-snowfight* package should also be installed using the following command in the console when your current directory and root directory are both *snowfight*:

```
pip install -e gym-snowfight
```

### *Human Gameplay*

After the installation, the game is accessible in a number of methods (all commands are typed while your current directory is *snowfight*):

```
python snowfight-starter/snowfight_play_v1.py -m 'human'
python snowfight-starter/snowfight_play_v2.py
```

Optionally, some arguments could be added (preferred):

| | | | | |
|---|---|---|---|---|
| `-ws` | or | `--window_size` | `16` | the size of the window. 16 is the default. |
| `-fps` | or | `--fps` | `40` | running fps of the game. 40 is the default. |

On the other hand, the random agent can be accessed by running the command:

```
python snowfight-starter/snowfight_play_v1.py -m 'human_rand'
```

### *Testing and Training of Models*

For testing the models, the following command should be runned:

```
python snowfight-starter/snowfight_qtest.py -f <file_path>
```

Note that, to visualize the testing in GUI mode, you can pass the argument:

```
-m 'human'
```

For training of models, the following command should be runned:

```
python snowfight-starter/snowfight_qtrain.py
```

Optionally, some arguments could be added to both *qtest* and *qtrain* (preferred):

| | | | | |
|---|---|---|---|---|
| `-e` | or | `--episodes` | `1000` | number of episodes to train. |
| `-ms` | or | `--max_steps` | `2000` | maximum game steps before force termination. |
| `-t` | or | `--n_threads` | `10` | number of threads used during training. default 6. |

[4] Pycharm environment is used by the developer(s).

If the outputs are desired, the following argument can be added:

| | |
|---|---|
| `-o <output.txt` or `.csv` or `.png>` | output the logs(.txt), data(.csv) or training graph(.png) |
| `-o <output_folder>` | outputs all of the above (including trained models in .h5 format) in the folder specified. |

Note that if the `-o` arguments are not specified, if you are training with `snowfight_qtrain.py`, then your files and logs will be automatically saved in the directory `data/qtable_<current time in YYYYMMDDhhmm format>`.

Inside the directory, training graph is stored in the filename `evaluation_graph.png`, while all best models will be stored in the filenames `model_best`, `model_best_replacedAt<episode>`. Please note that models will be stored in separate files *every 100 epochs*.

***Graph Plotting***

The training data of the models, saved in .csv format and generated by `snowfight_qtrain.py`, can also be read and further analyzed by using the command:

```
python snowfight-starter/plot.py -f '<file1.csv>,<file2.csv>,<…>' -o '<output_path.png>'
```