COSC 2527 Games and Artificial Intelligence Techniques Report

Semester 2, 2019

(Assignment 2 – Group 5)

Report Date	18/10/2019
Group Members	Mads Bjørn(s3799147) Haixiao Dai (s3678322) Yifan Wang(s3672150)

1. Decision Tree for Fences (by Haixiao Dai s3678322)

1.1 Overview

In Assignment 1 we let human to control fences go down and up through mouse click. In this assignment, we want to implement decision tree to let fences go up and down by themselves. When sheep are starving, the fences should go down to let sheep get out of the safe area to find fresh grass. After sheep go inside the safe area the fences should go up to prevent sheep from wolf. When wolf is close to the safe area where the sheep currently are, the fences should consider about the health point of the sheep to decide whether to keep the sheep inside or let sheep go. Also, when sheep come near by an area from outside, the fences should go down to let sheep go inside.

1.2 Objectives

For this part of the game we had the following core objectives:

- Observe users' behaviours, collect relevant data when user clicking fences and record data in a CSV file
- Using data collected from use to generate decision tree model
- Implement decision tree model to the game

1.3 Algorithm Selection

For this game, we choose J48 to generate decision trees. J48 is a subset of C4.5 and write in JAVA, embedded in WEKA. C4.5 was published by J. R Quinlan in 1993, by optimising and improve ID3 algorithm, "to prevent the emergence of classification evaluation function tend to choose more property values as the optimal attributes. C4.5 algorithm uses information gain ratio to replace the information gain as the classification evaluation function [1]." "C4.5 is the successor to ID3 and removed the restriction that features must be categorical by dynamically defining a discrete attribute (based on numerical variables) that partitions the continuous attribute value into a discrete set of intervals. C4.5 converts the trained trees (i.e. the output of the ID3 algorithm) into sets of if-then rules. This accuracy of each rule is then evaluated to determine the order in which they should be applied. Pruning is done by removing a rule's precondition if the accuracy of the rule improves without it". [2]

According to an article published on University of Regina, we choose C4.5 because of the following advantages:

- Avoiding overfitting the data
- Reduced error pruning

- Handling continuous attributes
- Handling attributes with differing costs [3]

1.4 Implementation

1. Collect Data

When user clicking a fence, the following data will be recorded: grass health point in current safe area surrounded by fences, distance between wolf and fence, distance between fence and nearest sheep, sheep position (inside, outside, middle), boss position (inside, outside), fence's current status ("up", "down")

2. Pre-clean data

Once we collected are the data we need, then we cleaned the data before using the data to generate decision tree. Because in the data, there were more records with "up" status than "down", to reduce the bias of the tree we randomly select both "down" and "up" records to make them have equal number of values. We also remove some irrelevant data (such as data recorded when user clicking the fence by mistake).

```
int count=0;
try {
    BufferedReader br=new BufferedReader(new FileReader("/Users/haixiao/Desktop/data.arff"));
    BufferedWriter writer = new BufferedWriter(new FileWriter(tempFile));
    String s;

    //Take out all records with up and down value for fence's status
    while((s=br.readLine())!=null) {
        lineread=lineread+1;
        if(s.contains("down")) {
            writer.append(s + System.getProperty("line.separator"));
        }
    }
    br.close();
    writer.flush();
    writer.close();
} catch (IOException e) {
    e.printStackTrace();
}

//Randomly pick null records
File tempFile2 = new File("/Users/haixiao/Desktop/dataclean2.arff");
try {
    BufferedWriter writer = new BufferedWriter(new FileWriter(tempFile2));
    String s;
    int nullcount=0;
    while(nullcount<120) {
        int lineread2=0;
        int randomnull=random.nextInt(390);
        System.out.println(randomnull);
        BufferedReader br=new BufferedReader(new FileReader("/Users/haixiao/Desktop/data.arff"));
    while((s=br.readLine())!=null) {
        lineread2=lineread2+1;
        if(lineread2==randomnull&&s.contains("up")) {
            writer.append(s + System.getProperty("line.separator"));
            nullcount=nullcount+1;
            br.close();
            break;
        }
```

3. Generate Tree

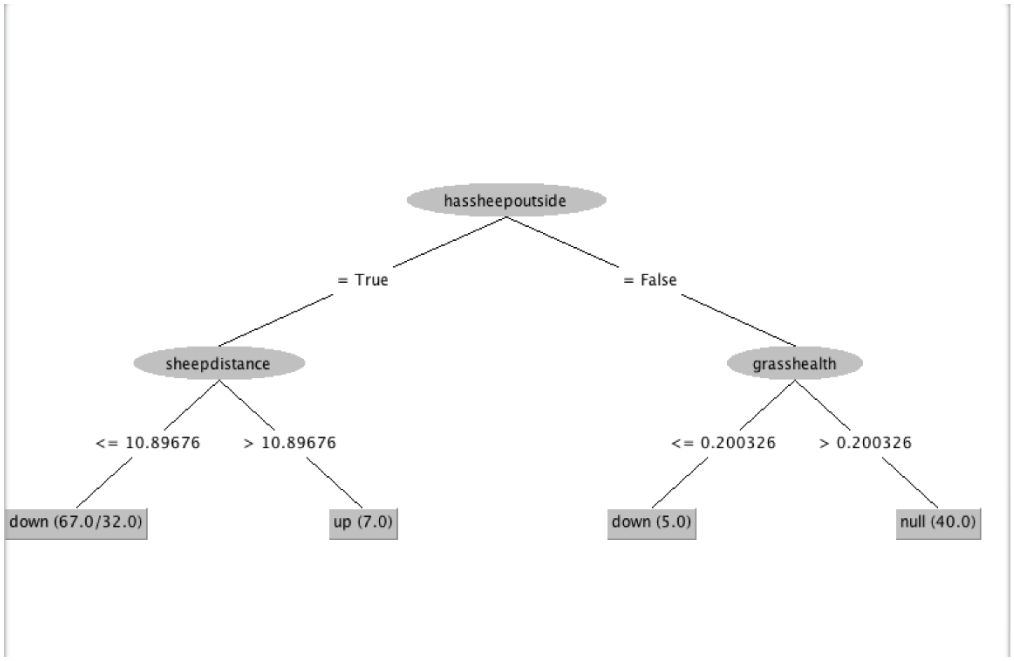
To generate decision tree, we choose WEKA as a tool to virtualise the result. We converted our CSV data to arff format, import arff to WEKA and WEKA will generate model and gives virtualised result. We tried two ways to generate the decision tree:

(1) Approach in early stage (first way):

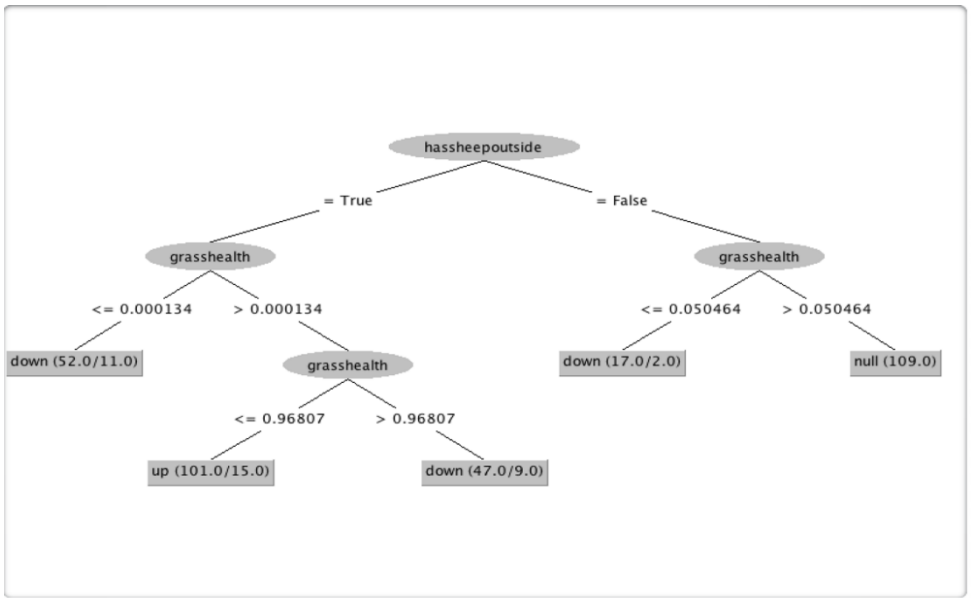
At the early stage, we recorded the fence's status as three value "up", "down" and null

Tree generated with 120 records (40 records each for fences status up, down, null):

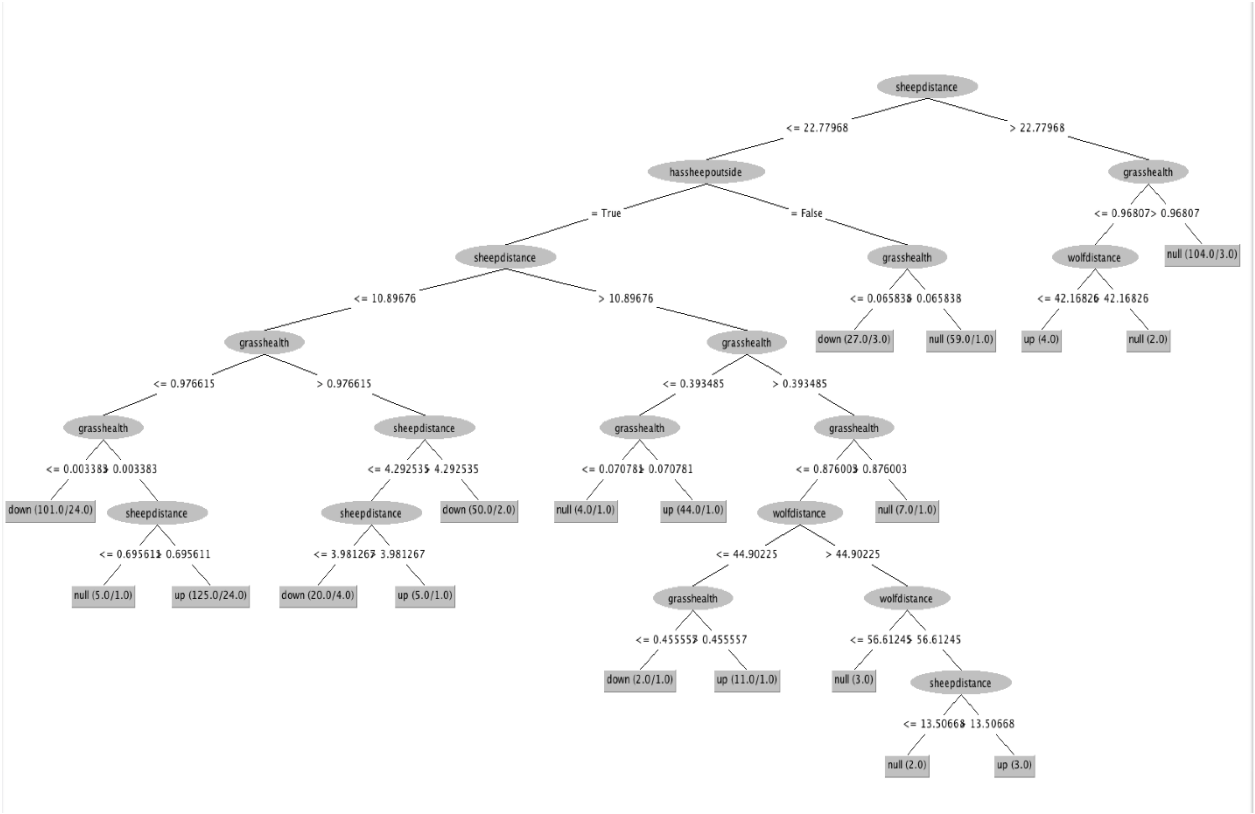
Correct Rate: 70.5882%



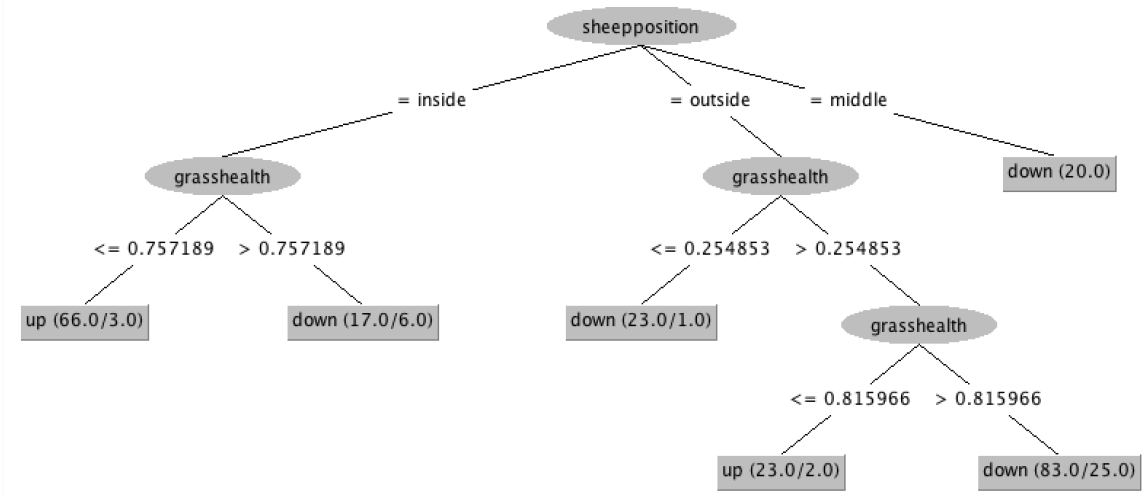
Tree generated with 300 records (100 records each for fences status up, down, null):
Correct Rate: 86.8098%



Tree generated with 600 records (200 records each for fences status up, down, null)
Correct rate: 83.218%



(2) Approach after optimisation (Sconed way):
After I found some problems during the first approach, I removed “null” for fence’s status, add bossposition when recording user’s behaviour and change “sheepoutside” to “sheepposition”.



Tree generated with 240 records (120 records each for fences status up, down), correct rate: 84.4828%

4. Tree selection

As I found some problems after generated decision tree with the first method and made some changes to the way I collect data before went through the second way, the tree generated in the second method seemed more realistic and matches with our expectation, so we choose the tree generated in the second way to implement in our game.

5. Implement decision tree to the game

For each leaf in the decision tree model, we use "if" condition to implement in the game.

Every time when frame updated, we will go through the tree once.

```
//Decision tree generated in the second method
public void decision2()
{
    if (bossposition.Equals("middle"))
    {
        fencedown();
    }
    else
    {
        if (grasshealth <= 0.15)
        {
            if (sheepposition.Equals("inside"))
            {
                if (grasshealth <= 0.08)
                {
                    fencedown();
                }
                else
                {
                    fenceup();
                }
            }
            else
            {
                fencedown();
            }
        }
        else
        {
            if (grasshealth <= 0.82)
            {
                if (sheepposition.Equals("inside"))
```

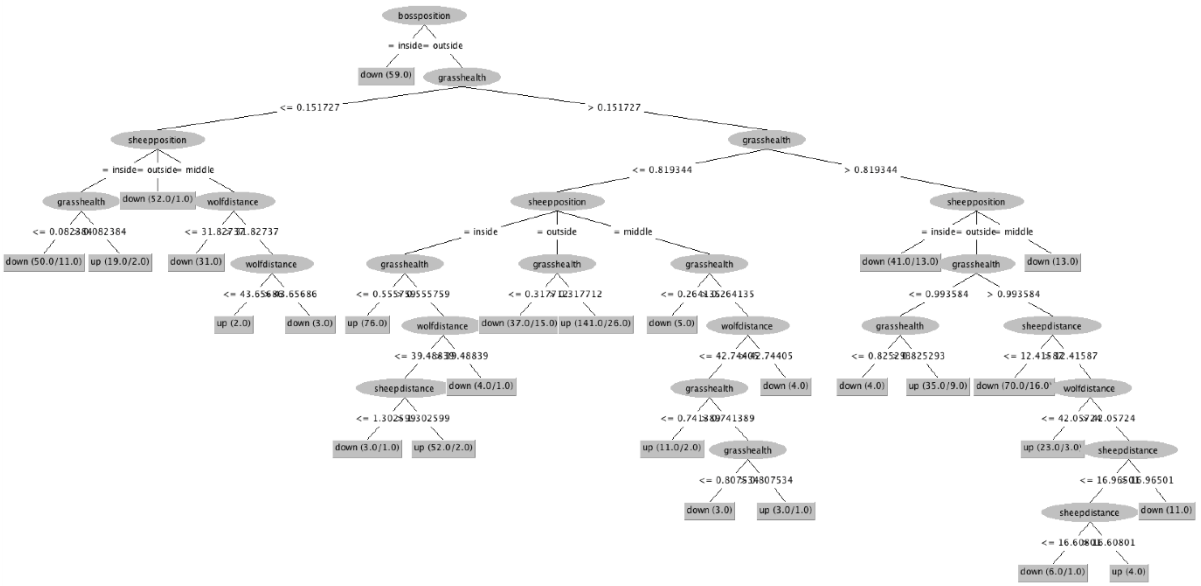
1.5 Challenge and problem solving

1. At the first stage, I only collected data when user clicks the fences and didn't pre clean the data to remove irrelevant records. As a result, I got a very big tree and the tree was not realistic to implement in the game. Then I realized if I only collect data when user clicking the fences, then the fences will keep truing up and down in every second. There were two ideas came out so solve this problem: First one was to collect data every second, if user click fences then I will record the fence's status as "up" or "down", if user do nothing then I will record fence's status as "null". The second option was to record fence's status as "up" or "down" in every second, if fence's status change from one to another that means user clicks the fence.
2. According to the tree, when sheep is inside the fences and the grass health point is less than 0.75 then the fences should keep staying in "up" position. However ideally when sheep are inside the fences and the grass health point becomes low the fences should come down to let sheep get out to find fresh grass. After exploring the data, I found the reason why this happened was when player playing the game, player didn't let sheep came out the fences unless the grass was completely died, and there was a delay which

the grass has completely died but player hasn't take any action to make the fences come down because I'm using gradient colour to represent the grass health and it will be difficult for player to make a decision on the exact time grass health becomes 0. So, in this case, when the grass health becomes 0 and sheep are inside the fences area, the player won't take any action and the data will still be recorded. As the "down" status lasted very short so although I had same number for "up" and "down" record but the number of records with grass health less than 0.1 along with sheep position inside and status down were more than records with grass health less than 0.1 with sheep position inside and status "up", so WEKA might consider the second type of record as fault value and used first type of record to generated the tree. To solve this problem, I wrote a JAVA script to remove those records which caused by the reaction time by the player and finally the tree became more reasonable.

```
if(Float.valueOf(split[0])<0.1&&split[3].equals("inside")&&split[5].equals("up")) {
    continue;
}
if(lineread2==randomnull&&s.contains("up")) {
```

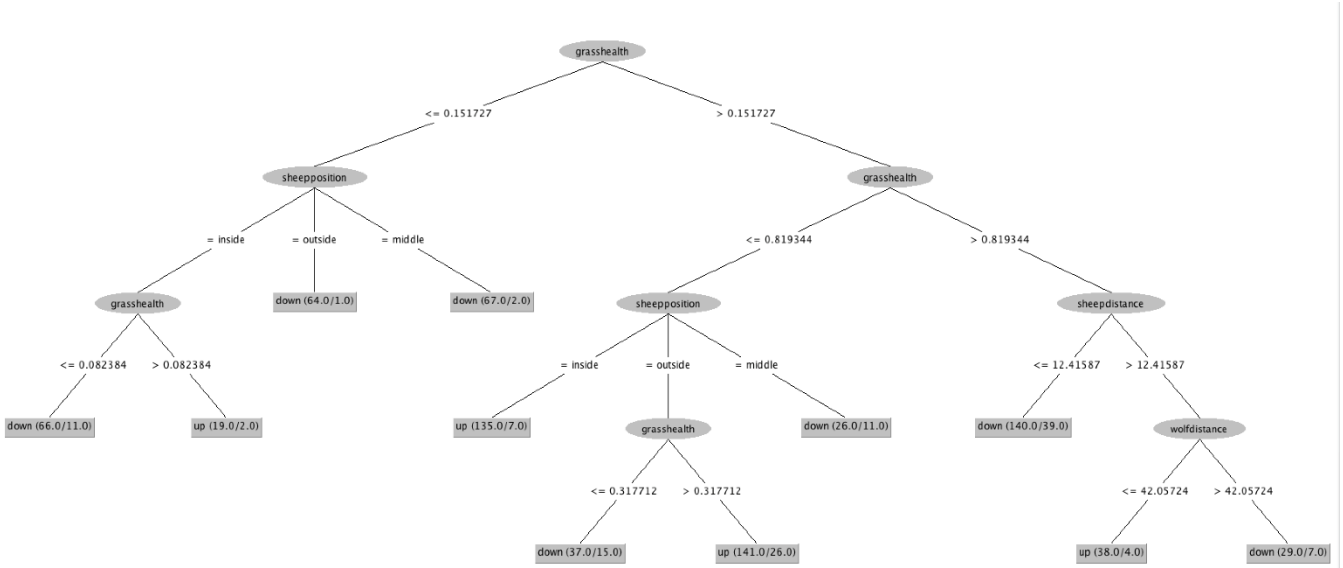
3. At the beginning I used the default configuration of WEKA J48 to generate the tree. However, the tree generated at the first time was too big and not very realistic to implement to the game.



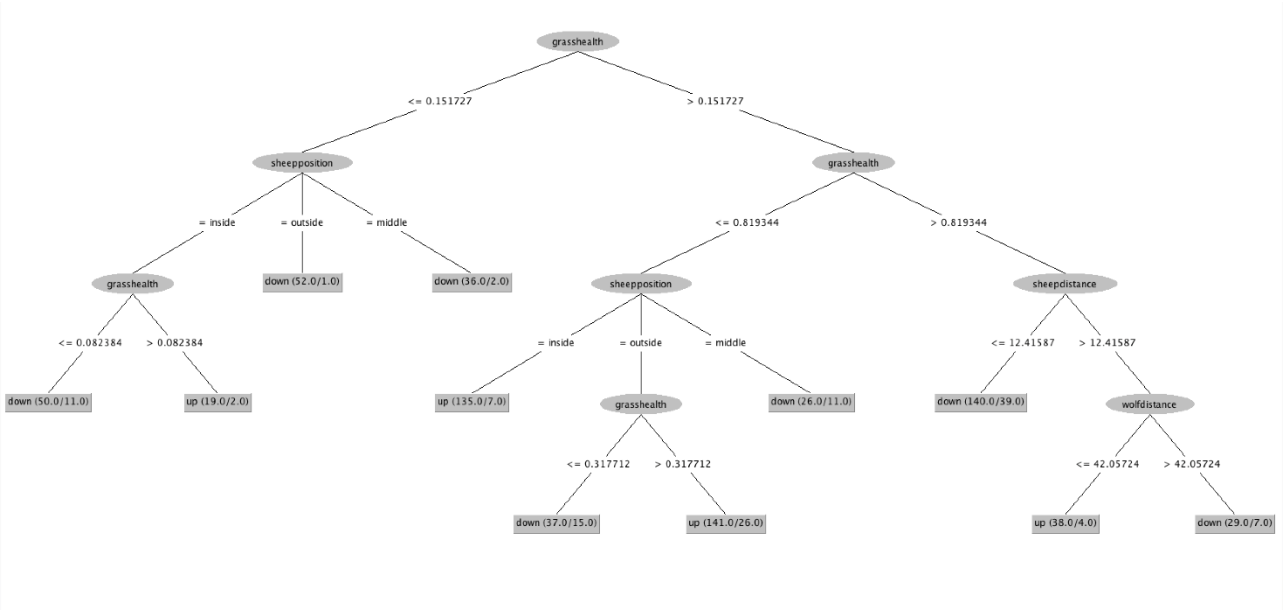
Correct rate: 81.5%

After explored the tree I found some leaves had very small value (e.g. the number of "up" on the bottom-right is 4) and those leaves seemed not quite reasonable so I think those leaves might be generated by unrelative records which caused by player clicking fences by mistake. Then I tried to increase the minNumObj (minimum number of instances per leaf) in WEKA configuration to solve this problem. However, after I increased this value

to 5->10->15, the tree became better, but the only problem was the "bossposition" value was missing after I increased the minNumObj (which was on the top of the first tree). However, ideally the fences should keep in down position if the boss is inside the fences because in this case the boss is pushing sheep out the fences and this behaviour shouldn't be affected by other values. After I increased the minNumObj value the tree was like this:



I think the "bossposition" was missing because the records were not enough to find a relation between bossposition and fence's status as the number of the records which can represent this relation was less than the minimum number of instances per leaf I set. And if I record more data, I'm worry about the tree might getting bigger and I need to increase minNumObj value again, as a result I might still not be able to get the "bossposition" counted in the tree. I also tried to reduce the number of data to 240 (120 each for up/down) and decreased the minNumObj value but the bossposition was still missing. To solve this problem, I combined the first tree with the second tree. I took the “bossposition” leaf from the first tree, then remove the 59 records related to this leaf to generate the second tree and use the second tree as “else” leaves in the first tree. To implement in the game added an if condition to check the bossposition first, if the bossposition==inside then go “down”, else go to the decision related to the second tree. After removing those records, the tree was like this:



Correct rate: 78.95%

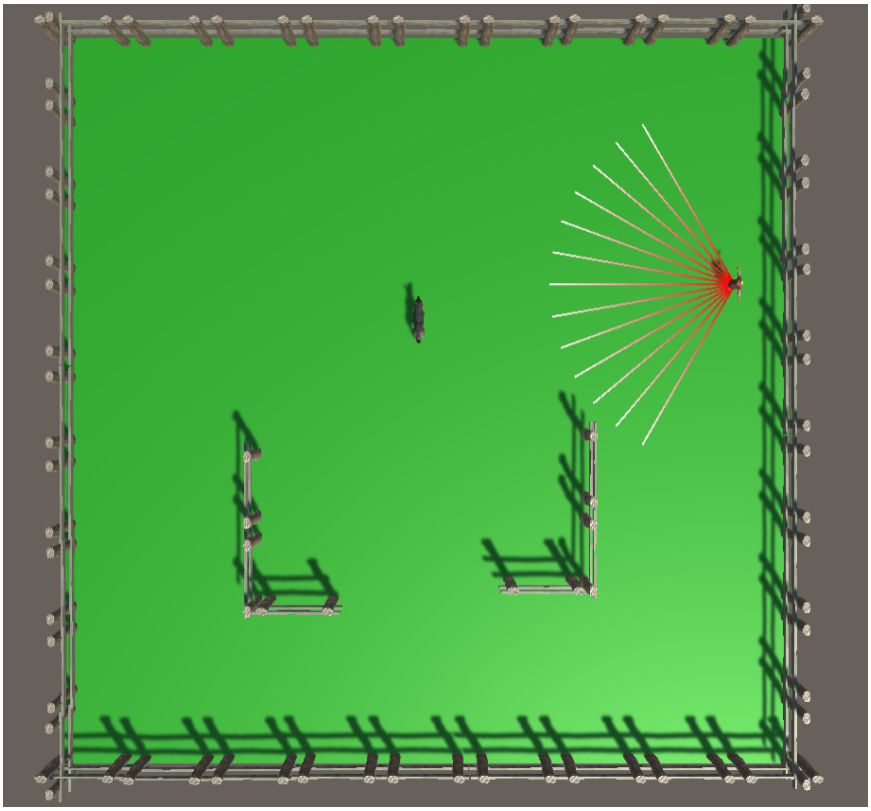
This tree was the final decision tree we choose to implement in our game.

2. Deep Reinforcement Learning for Wolf

(by Yifan Wang S3672150)

2.1 Design and Intention

For the first assignment, our wolf uses path-finding to find the target sheep, and to avoid obstacles. For avoiding the farmer, we use FSM to control the different situations. There are a lot of things we need to manage, and path-finding also need a lot of calculations, sometimes it does not work good. Therefore, we want to design a complex environment and using machine learning to control wolf's behaviour. Our intention is that training a wolf can find sheep, avoid fences, and avoid farmers intelligently in the complex environment. At the beginning, we designed 3 patrol farmers, random fences like a maze. But we quickly found it needs lots of time to training the wolf, it would take much more time than we expected. Finally, we changed the environment to a simple environment with only 1 patrol farmer, and 2 fences as obstacles.



2.2 Algorithm choice

As our team members have used decision tree algorithm for fence the and q-learning for the farmer. And I just hear about Unity ML-Agents from another course (mixed-reality), but I never use it before. After I read the documents and learn some examples from git, I decide to use ML-Agents for the wolf training. Because our wolf game is suitable for Reinforcement Learning and Neural Networks which equals Deep Reinforcement Learning, and in ML-Agents, there have a modularized way to training by Deep Reinforcement Learning. Since our wolf only have 2 actions, moving forward/backward, rotate left/right. We can easily give the wolf a feedback(reward value) after it did one action to decide either the action is good or bad. There are only three situations in our game: eat a sheep(good), collision with farmer(bad), collision with fence(bad). So, it is suitable to using Deep Reinforcement Learning to train our wolf.

2.3 Implementation

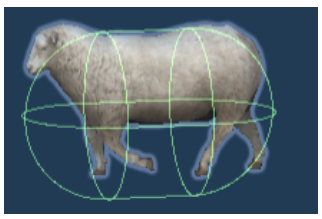
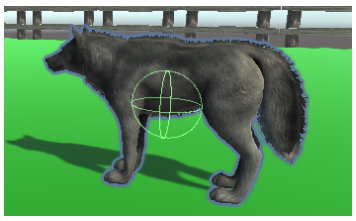
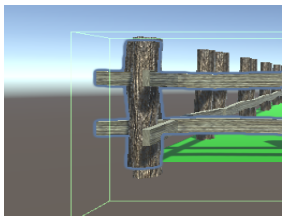
2.3.1 Implementation

First, import ML-Agents Toolkit from git.

Second, create our environment.

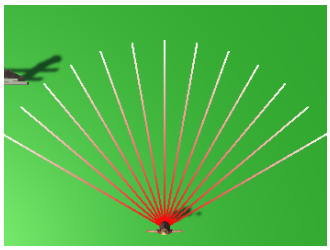
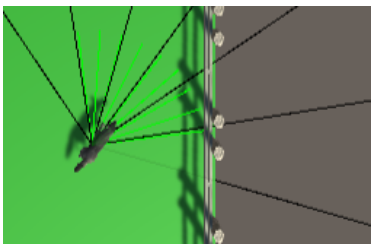
Because we already done a game in assignment 1, so we just use the prefab from our assignment1. For assignment 2, we have 3 characters: wolf, sheep and farmer. We want the sheep can be killed by wolf as well as wolf

killed by fence, using collision, so we need to add collider to both sheep, wolf and fences.

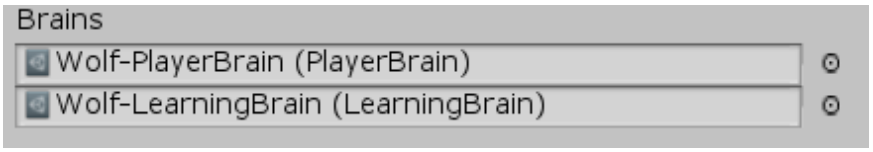
Also, we want wolf to move forward/backward, rotate left/right. We add rigidbody for wolf.

We want the partol farmer to detect the wolf, we add ray cast for the farmer. We also add raycast for the wolf help it to detect the environment.

After we finished the environment, we will using ML-Agents to training our wolf.

To using ML-Agents, first we create two brains and added them into academy.



The learning brain is for us to train the wolf, and the player brain is for us to test the environment and actions before training.

Then we create a script override the agent script, called WolfAgent. This will be our core part of using ML-Agents.

In WolfAgent script, we need to define the Reset() function, this will be run after calling Done() and at the start of training (game play). We want to reset the wolf position and rotation randomly when wolf collision with fence or be ray hit by farmer.

After that, we need to determine what observations we need to feedback. In this game, we using the ray(hit.distance) to find the distance

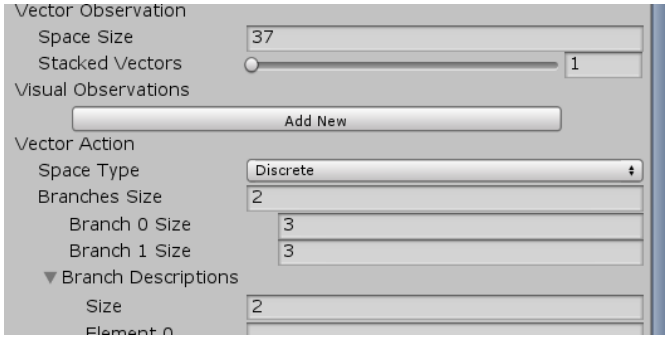
between wolf and other gameobjects and using tag to differentiate sheep, farmer and fence. Also, we need to add the wolf's velocity at x- direction and z- direction.

Then we add the 2 actions we talked in algorithm choice, move wolf forward/backward and rotate wolf left/right in override AgentAction() function.

Finally, we should add reward for different situation, when wolf eat a sheep, AddReward(1), when wolf collision with farmer or fence, SetReward(-1) and reset the training. These reward values can be modified by different training model, but these are best for our game after our lots of experiment. Also, we do not have to call AddRewad() in AgentAction() function, we can use AddRewrd() at every update() function, like onCollisionEnter(), onCollisiionExit() and etc.

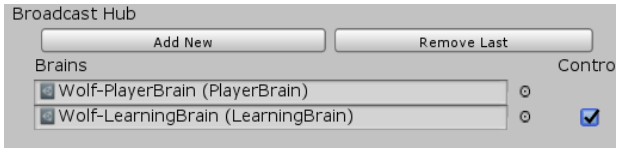
After we did a lot of stuff, we almost ready to start training our wolf.

We need to set our space size as we defined in AddObservations() function before. And set action branches size as we defined in AgenetAction() function before. For this time, we use Discrete space type. Doing above thing for both learning brain and player brain.



And for player brain, we need add more player actions to control the agent(wolf).

Now we can test our environment by using player brain. If there is not any error, we change the player brain to learning brain. And select the control box.



Finally, we add random walking script for the sheep, and patrol script for the farmer. And we build the project.

Now we can use python to train our model. We select Proximal Policy Optimization(PPO) – the default trainer_config to train.

For different models, we need change the batch_size, buffer_size to make the training result better.

Batch_size: number of experiences used for one iteration of a gradient descent update. (We use 64)

Buffer_size: how many experiences should be collected before updating model. (we use 6400)

And Max_step for determining max steps we want to train our model. (We use 7.0e5)

Then it will be taking a long time to wait the training result. (Maybe one hour, two hours, three hours or longer.

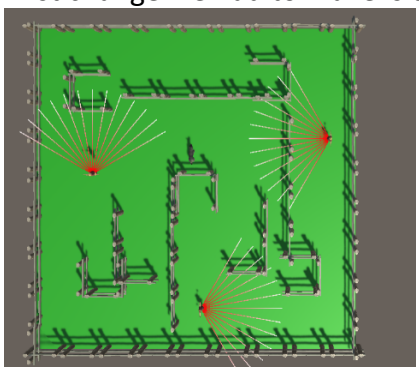
After training, we can use below commands to get the graphs about the training.

```
tensorboard --logdir=summaries
```

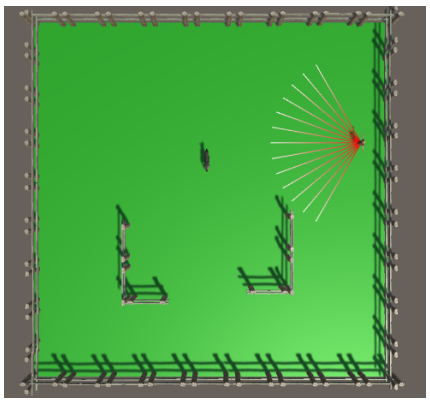
nd then opening the URL: localhost:6006.

2.3.2 Changes

First change we had to make is to change the environment.



Old environment.



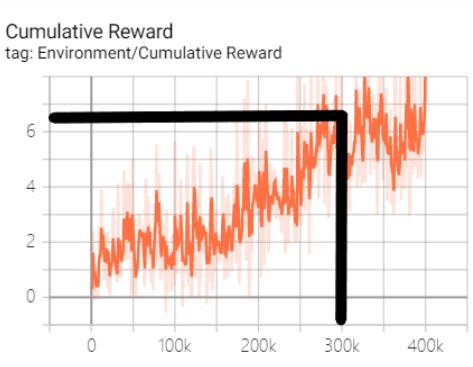
New environment.

As we discuss at beginning, complex environment took our much more time to train than we expected. So, we change to a simple environment.

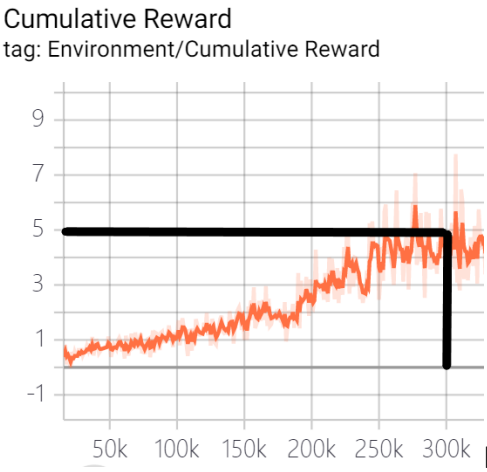
Second change is we change the way the wolf eat sheep, at beginning , we wanted to use collision, but both wolf collider and sheep collider are too small. It may take long time for train. So, we add a green ray to the wolf, if the sheep hit by the ray, it will die. This change also saves us lots of time.

Other changes are all about data. We need to try a lot of times to find the best way to add reward and set batch size and buffer size to make the training result better.

Here is an example how different buffer size leads to different training results.



buffer size: 1000



buffer size: 6400

As we can see from above graphs, the smaller buffer size – faster training updates but less stable, the larger buffer size - more stable training updates. Finally we chose the larger buffer size to make our training result stable.

2.3.3 Problems

The hardest problem we have met is that we train many times, the reward always grows faster at beginning but stop grow at end, and the reward may be lower than at beginning. We try a lot of ways, for example, change how to add reward(Addreward), change the action input, change the observations, but they all don't work. Finally, we read the documents of training PPO(ML_Agents) on the github carefully, and find out that for our

environment, better select small batch size and large buffer size. It really help us to get good stable reward while training.

Another problem is the way we move wolf agent, in unity, we usually use

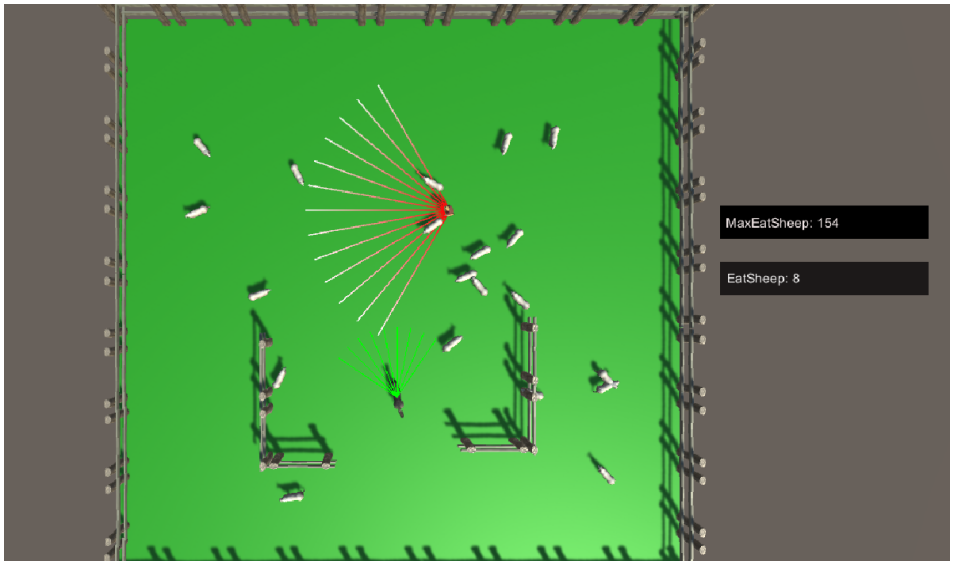
transform.Translate(Vector3.forward * Time.deltaTime * speed*actionSpeed);

to move gameobject move forward. But this causes a bad result, it's hard to collect the observation of speed for the wolf agent. And finally we change the way, we use

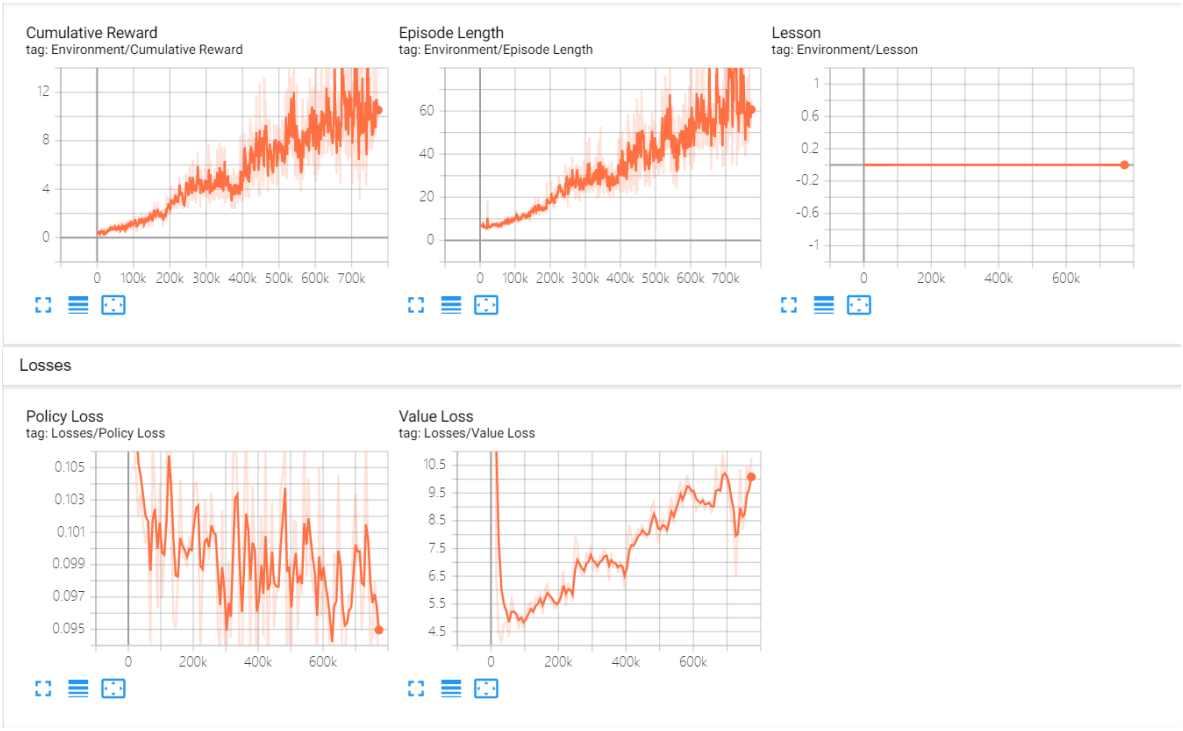
rBody.AddForce(dirToGo * speed, ForceMode.VelocityChange);

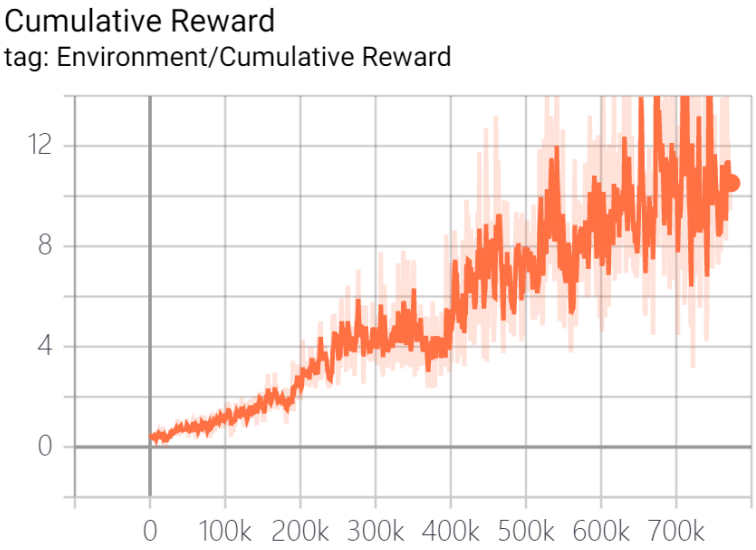
to solve this problem. For observation, we can easily add ***rigidbody.velocity.x*** and ***rigidbody.velocity.z***.

2.4 Outcome



We are satisfying for the training result. The wolf looks smart. It can even eat 154 sheep using only 1 life. And the wolf knows how to avoid patrol farmer, avoid fences and wall. One unexcepted behaviour is that wolf knows to go the area with more sheep. I think this may be the collection of observations collected all ray(hit.distance) for all sheep, and the wolf learn to go to the more sheep area to get more rewards.





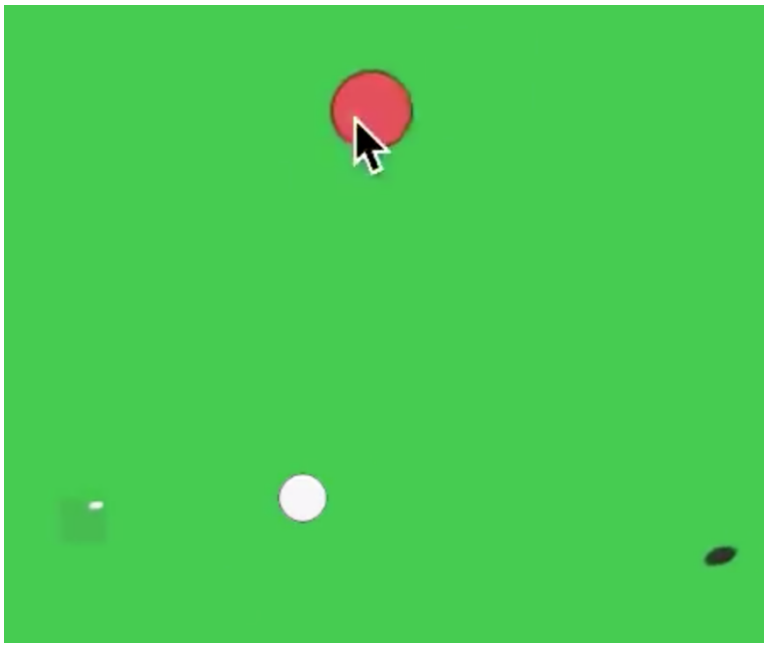
The above graphs show the reward is grow stable while we continue training. But we don't have enough time, the average reward after 700k steps is 12. If we training more time, the wolf agent will behave smarter.

3. Q-learning for 2D Game (Made by Mads Bjørn s3799147):

3.1 Introduction:

For the new 2D game the graphics is quite basic, this is because the time was used on training our Q-table, and because we the game is rebuilt from the bottom up, with no reuse from our old 3D game. 2D game still have all the features needed from assignment 1, with full steering, flocking and A* pathfinding, again no reuse from the 3D game since we could not even use Unity's built in vector library.

The characters in out 2D game is:



The little oval white thing in the bottom left corner, is a sheep.
The black oval in the bottom left corner is the wolf.
The red dot is the farmer.
The white dot in the bottom middle is the helper. This helper does the exact same as the farmer, except it is controlled by the game.

In normal play the farmer is controlled by the arrow keys, but the game is quite easy with the helper, so we left the farmer controlled by the mouse to make it easier to play around with.

3.2 Where to change what:

The game is played from the main scene in the scenes folder.
To play with a specific helper (trained on 1, 2, 3 or 4 sheep), go the Assets/EngineRunner.cs and change the number of sheep on line 16.

For training only, the files inside Assets/G5Engine/ is needed, just run the Driver.cs after setting the settings.
All videos referred to about the 2D game, is in the root folder of the zip file with the 2d game.

This game is played out by us controlling the farmer, the farmers goal is to keep the sheep alive for as long as possible. The sheep is flocking and eating the grass where it is currently positioned. If the sheep eat all the grass where it stands it will die. If the wolf gets a sheep, the sheep will also die. In this game we want to have 1 helper, which is controlled by the game. The game is played out on an infinite large world.

Before starting to program this 2D game, I realised that to train this I probably needed this to train faster that unity could run the game. Therefore, I build the 2D game from bottom op, so no parts are reused from the old 3D game.
I wanted to be able to train this on a server without Unity, therefore I build the game without any Unity libraries. This is also why I use my own vector class, to do vector math.

3.3 The sheep:

The sheep use boids[4] flocking except we do not have any alignment, since we learned from assignment 1 that alignment in sheep looks odd. The steering of the sheep is made in the same way as in the old 3D game, with dampening on both the angular velocity and linear velocity, when the sheep approach its target.

3.4 The grass:

Each grass patch is 1 by 1 meter and provides food for one sheep for 10 seconds. If no sheep is on the patch, then it takes 3 seconds to get fresh and ready again. Since we play in an infinite large world, we store the grass information in a Dictionary, where only the grass patches with less the full health is stored. In this way we can store only the needed information and know that a grass patch is at full health if it is not in the Dictionary.

3.5 The wolf:

The wolf will try to get close to and kill sheep, if the wolf gets within 1.5 meters of a sheep, the sheep dies.
The wolf uses A star pathfinding:

This runs a grid of 1 by 1 meters

The only obstacles we plan around is a 4-meter radius around any helpers / farmer.

This is a multi-goals A star, where the heuristic function is the Euclidean distance to the nearest sheep.

To make sure the training can run fast, the A star is limited to 1000 iterations per search. If the A star goes over this limit, the wolf will teleport to a sheep with a helper nearby. We make sure that there is a helper nearby and it teleports 20 meters away from the sheep, to give the helper time to react.

The wolf replans a path once every second, and in the intermediates updates it follows the already plan path. It does this in the same ways is the old 3D game, by applying a force

towards the furthest point in the plan that we have a line of sight to. (Without getting closer than 4 meters to any helper)

The steering is the same as with the sheep.

If the wolf gets closer than 4 meters to any helper it goes into running away mode, in which it will stay until it is more than 15 meters away from all helpers.

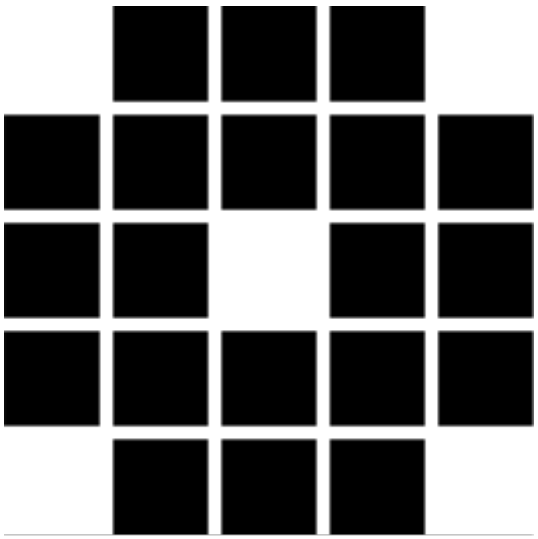
If all sheep are within 4 meters of a helper, the wolf will come to a standstill since it cannot plan any path without coming closer than 4 meters to a helper. This is on purpose, and the wolf will continue whenever the sheep runs away from the helper.

3.6 The helper:

The helpers that is controlled by computer have 5 actions, (up, down, left, right and stand still) each of these actions is applied for 1 second before a new action is chosen. This is also the way we make a continuous game into a discrete time step.

3.7 State encoding:

To encode states, we use a local grid centred on the helper. This grid is made up of 20 states and is illustrated below:



Each of these grid squares are 3.5 by 3.5 meters. And will have 1 of 3 values:

1: No sheep
2: One or more sheep that is on a square(s) with plenty of grass
3: One or more sheep where at least one is on a square with little grass left.

Here there is no information about the state of the grass when no sheep are on a square. Since grass regenerates in only 3 seconds, we do not lose that much information. But we get the state space down from 4^{20} to 3^{20}.
To further reduce the state space, I limited the numbers of sheep:

E.g. if we have 4 sheep in each of their own square and 3 of them have lots of grass with 1 almost out of grass, we can calculate the number of combinations as follows:

$$\frac{20!}{(20-3-1)!*3!*1!} = 19380$$

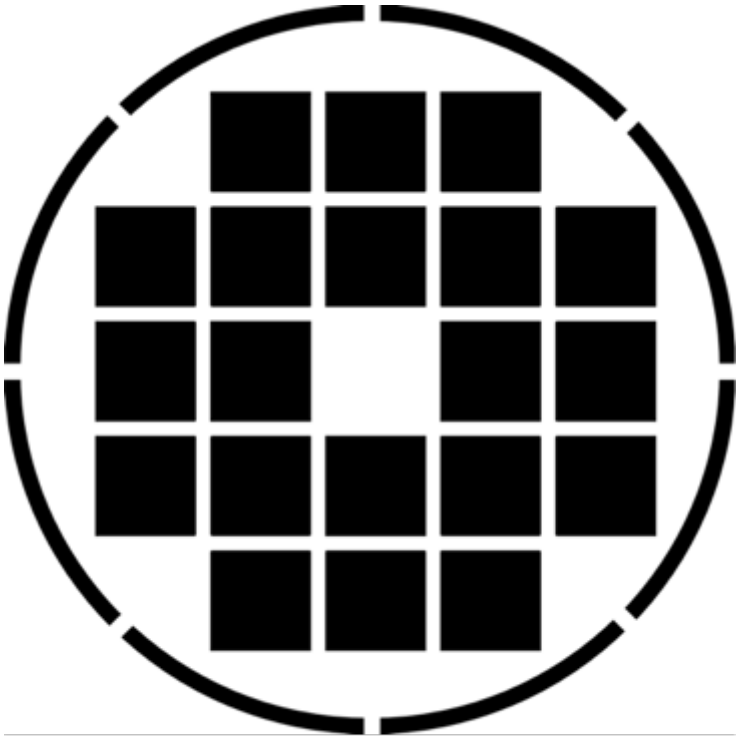
But with 4 sheep we must consider all combinations of on low grass and not:

$$\sum_{k=0}^{4} \frac{20!}{(20-(4-k)-k)!*(4-k)!*(k)!}$$

We also need to consider that not all of the 4 sheep are in a square, but some of them could be out of range:

$$\sum_{x=0}^{4} \sum_{k=0}^{x} \frac{20!}{(20-(x-k)-k)!*(x-k)!*(k)!}$$

We also need to encode the position of the wolf; this is done by saving the wolf's position as one of 28 values. These 28 values are illustrated below:



The 20 squares the same as before, but with the 8-ring segment. A segment is used if the wolf is out of range, but still in that general direction.

This gives 28 new states per state we had before:

$$\sum_{x=0}^{4} \sum_{k=0}^{x} \frac{20!}{(20-(x-k)-k)!*(x-k)!*(k)!} * (20+8)$$

Number of sheep	Theoretical max number of states
1	1 148
2	22 428

3	277 788
4	2 448 348

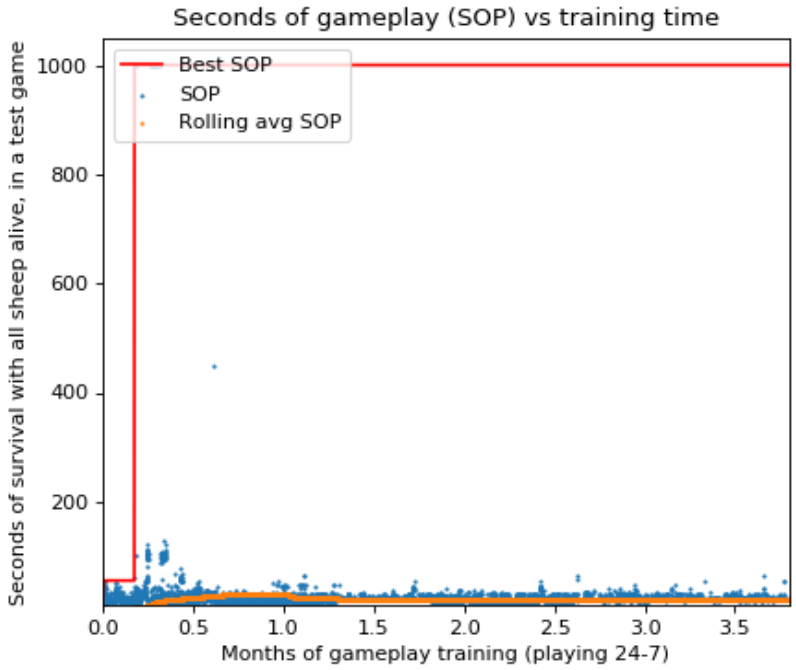
A simple Q-learner class was implemented since the math is easy enough when we only must support a Q-table. Though this Q table was implemented as a Dictionary, so we only store the states we have explored. [5][6]
When learning we found, through trial and error, that a learning rate of 0.025, and an exploration rate of 0.1 worked best for learning the Q table.

When we simulate the game, each second of gameplay consists of 60 small updates.
When training we can simulate 10 000 000 seconds of gameplay in just under 1 hour.
Meaning that we can simulate at 2 800 times normal speed. Therefore, proving it was a good idea to implement the game engine separated from Unity.
This also means we can train our Q-table for what is equal to years of real time gameplay.

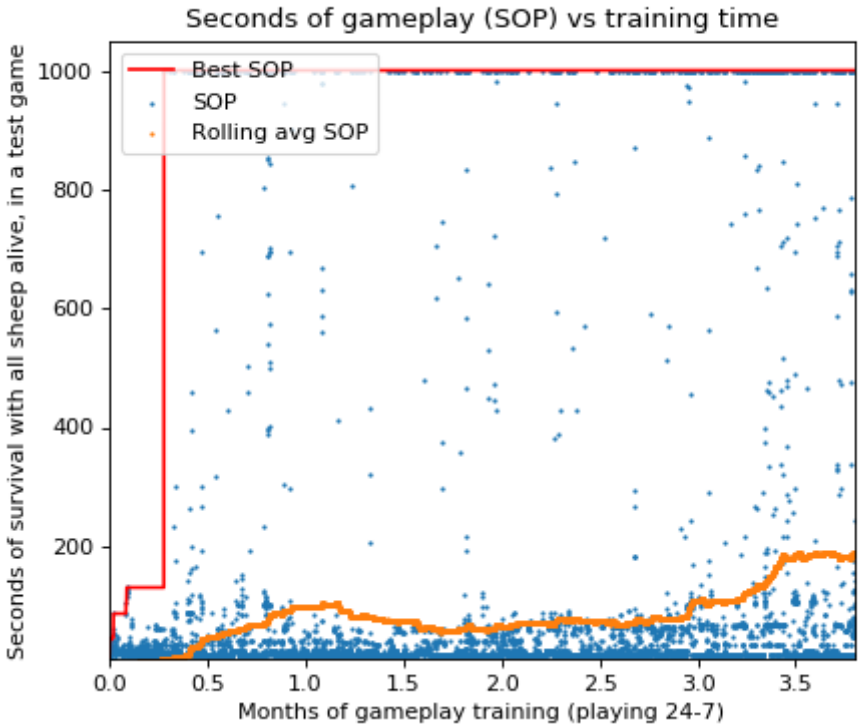
3.8 Reward and discount factor:

To find the best reward and discount factor, we tried different values for each to see which would get the best result. For this training we only train with one sheep and give negative 1 for losing the sheep. We then give 0.001, 0.01 or 0.05 for surviving each second (per action). We also tried 3 different discount factors of 0.6, 0.95, and 0.99:
All 9 combinations is show below:

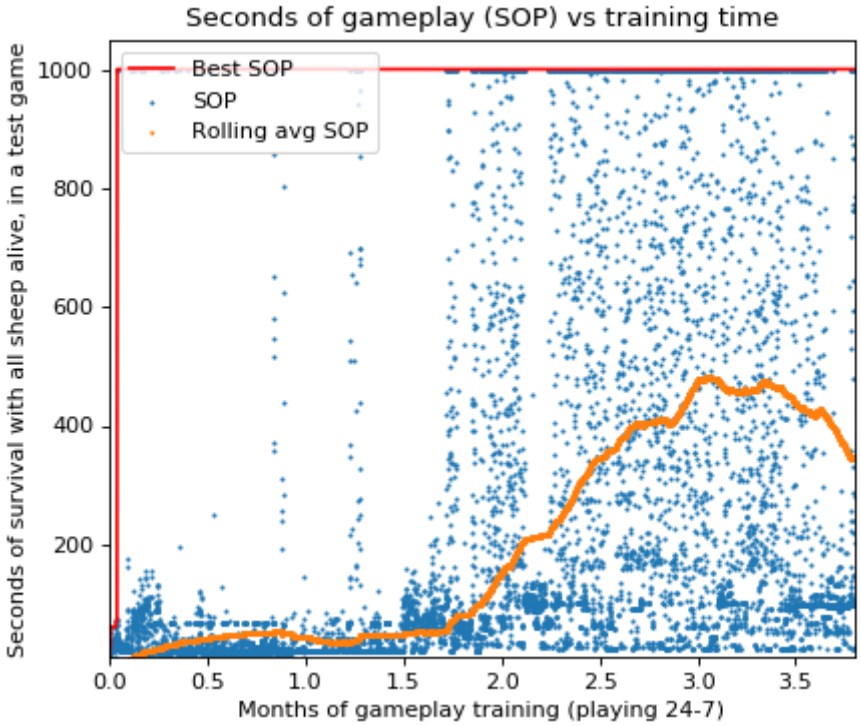
Discount factor 0.60 and positive reward per second 0.001:



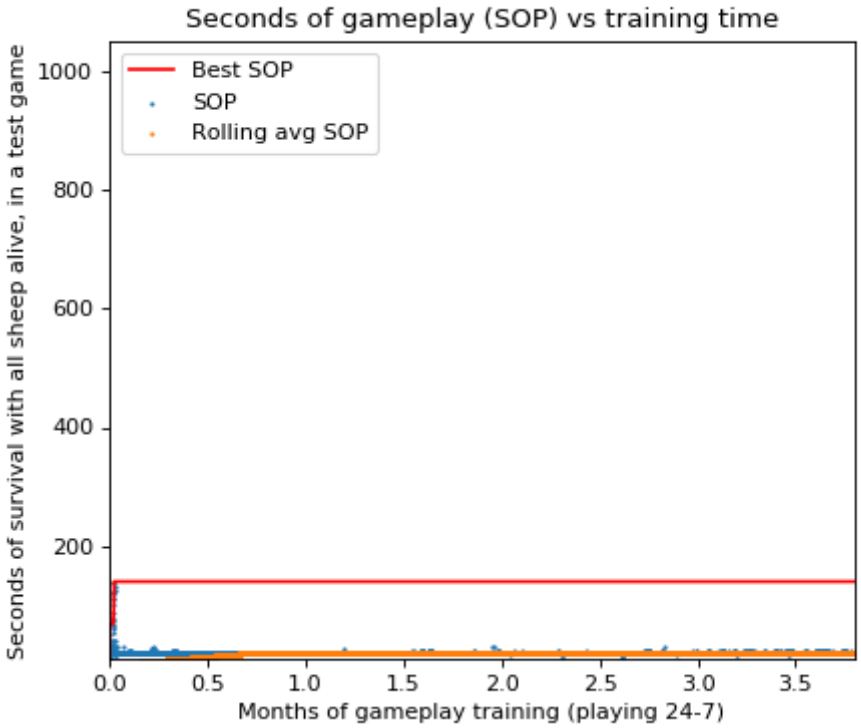
Discount factor 0.60 and positive reward per second 0.01:



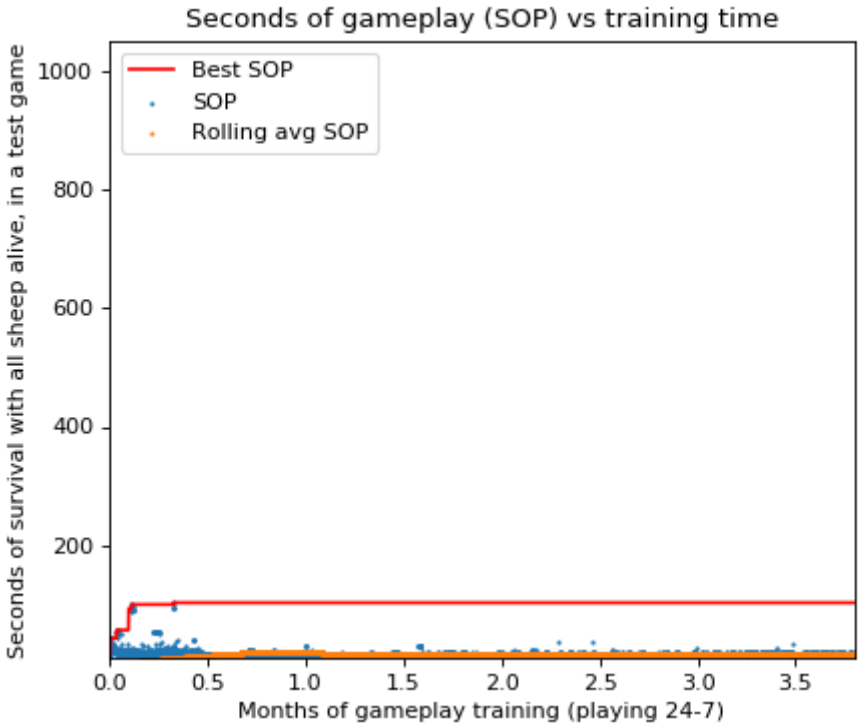
Discount factor 0.60 and positive reward per second 0.05:



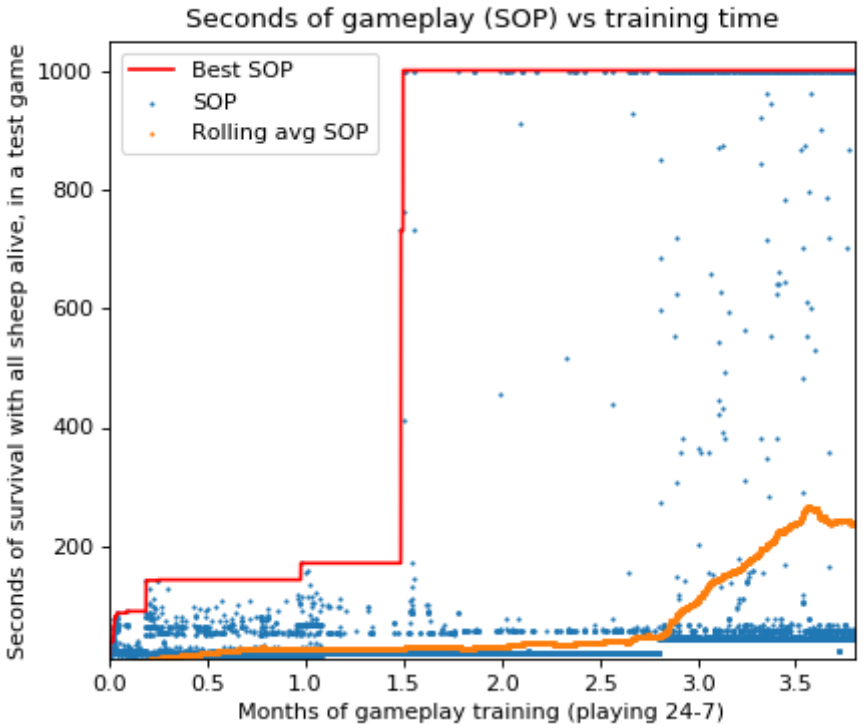
Discount factor 0.95 and positive reward per second 0.001:



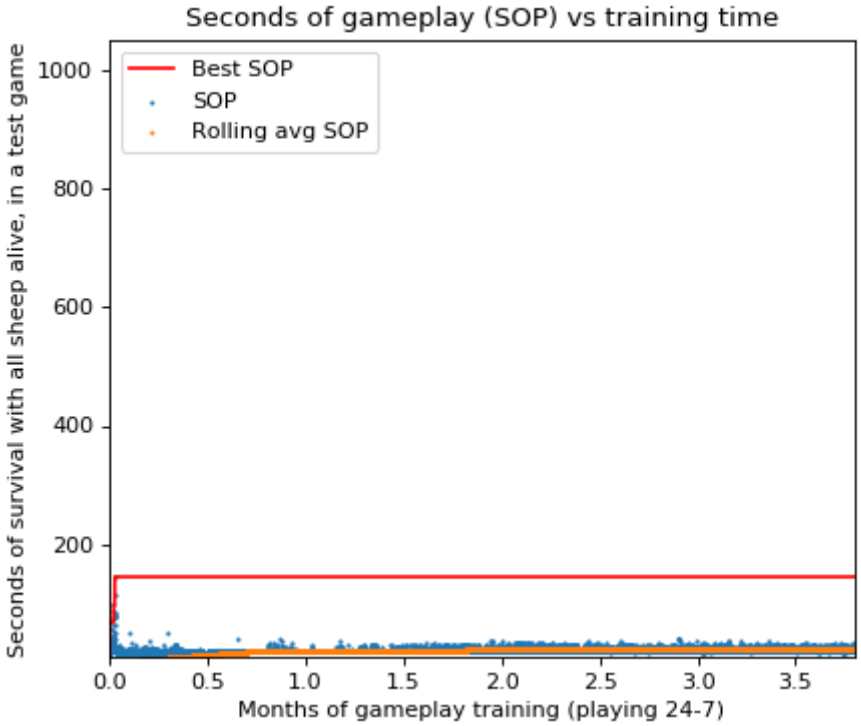
Discount factor 0.95 and positive reward per second 0.01:



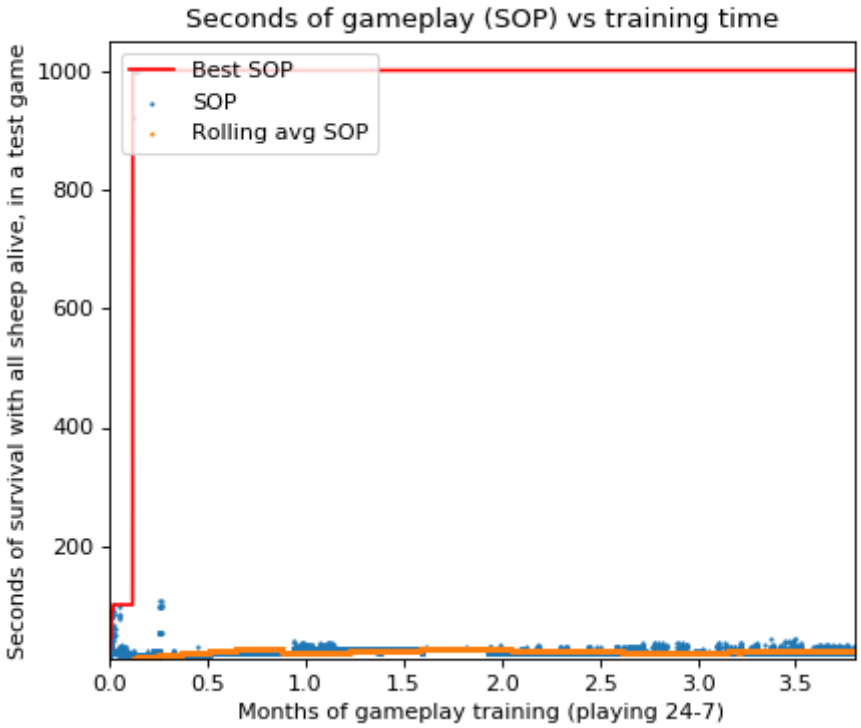
Discount factor 0.95 and positive reward per second 0.05:



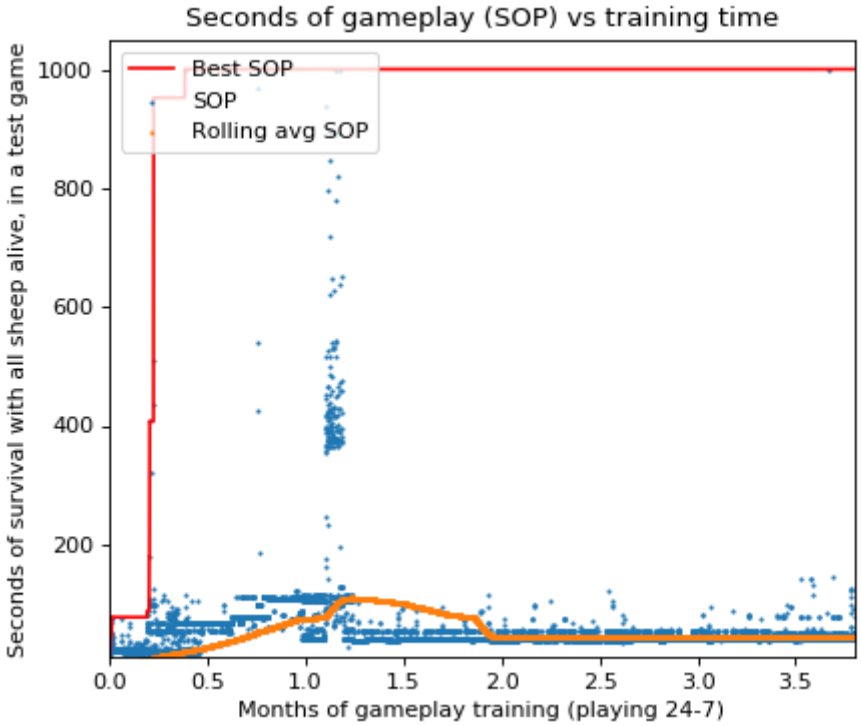
Discount factor 0.99 and positive reward per second 0.001:



Discount factor 0.99 and positive reward per second 0.01:



Discount factor 0.99 and positive reward per second 0.05:

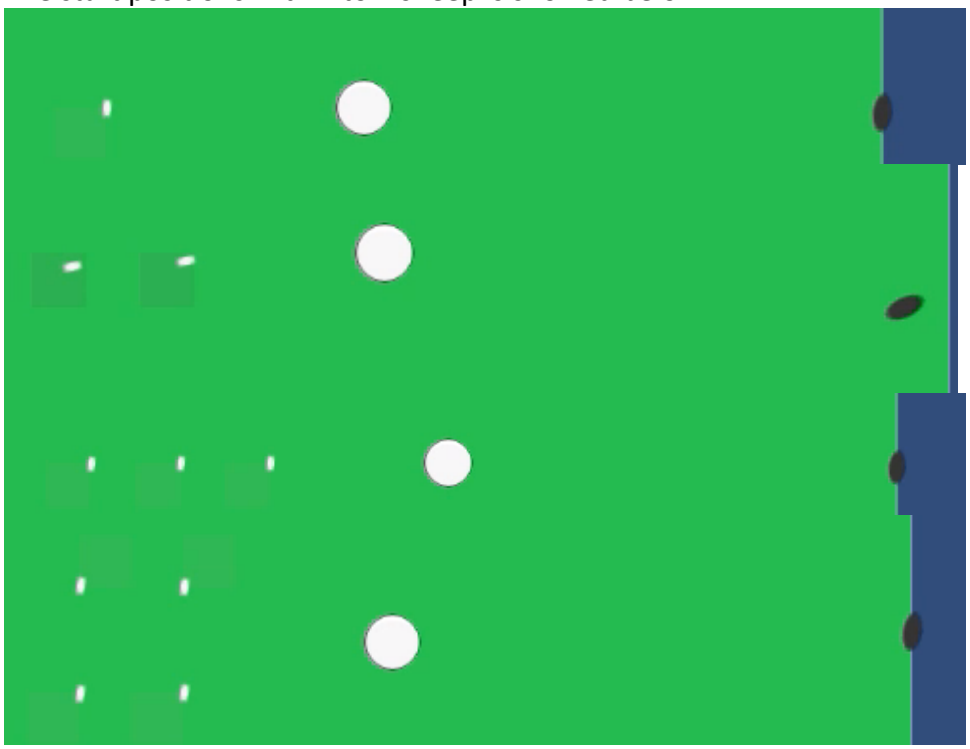


From this we can see that it clearly gives the best results by using a discount factor of 0.60, and 0.05 reward per second alive. Therefore, we will continue with these settings throughout the

3.9 Results

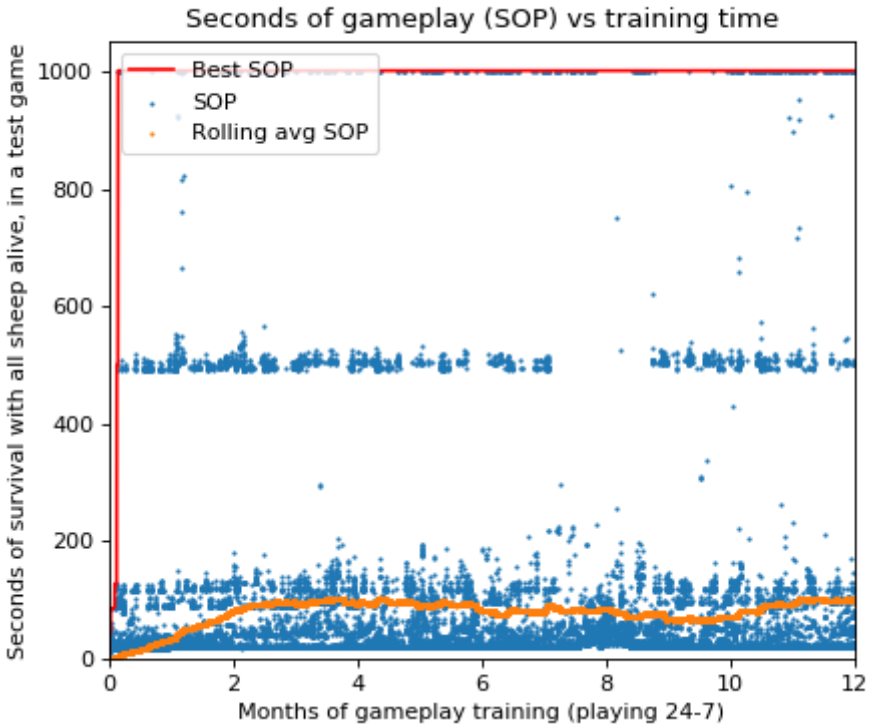
When training with multiple sheep we will count it as loosing when you lose your first sheep. This is done since getting good results with one sheep is easy, and we want to avoid the helper just losing all sheep except one and then just continuing with only that sheep.

The start positions with 1 to 4 sheep is showed below:

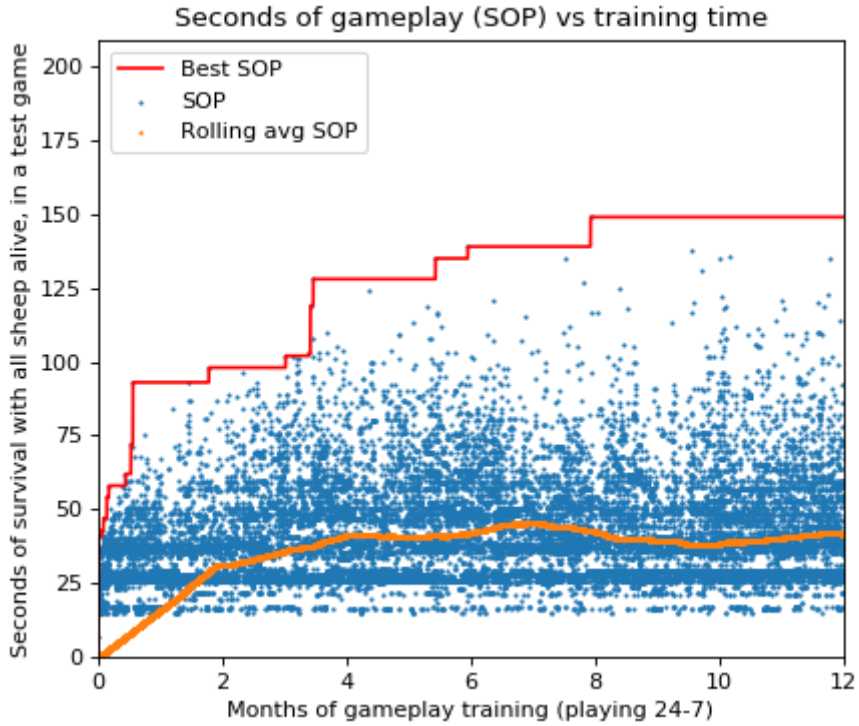


Below are the training results for the different amount of sheep (Note we have training for longer that what is shown on the chart, but no significant improvement came of it):

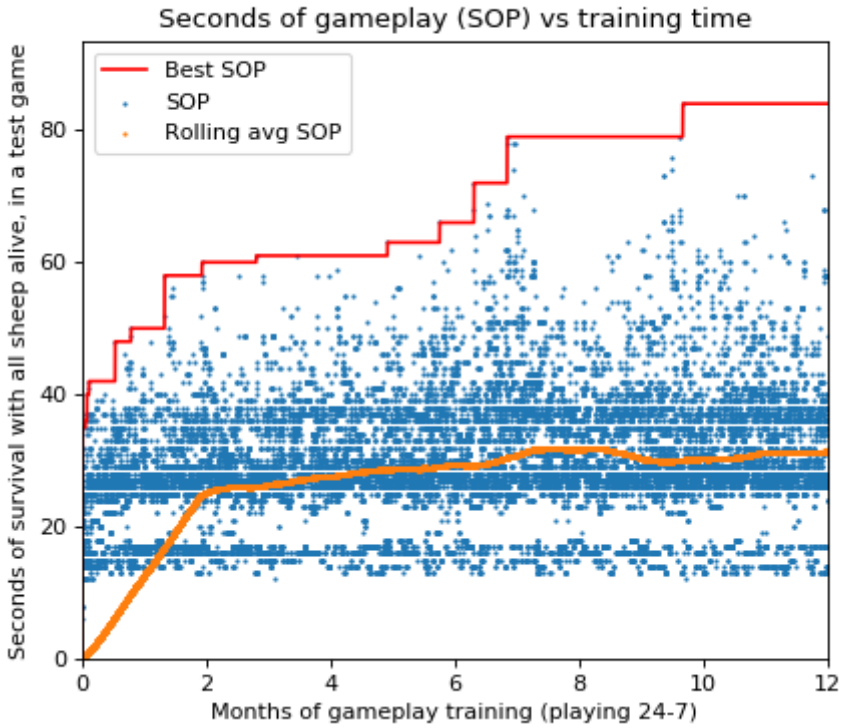
With 1 sheep:



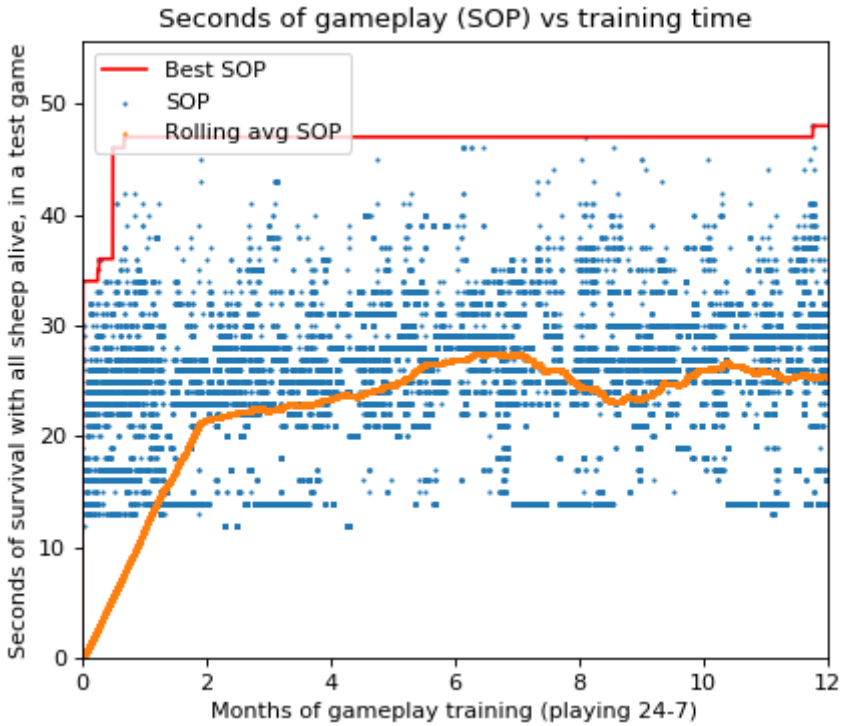
With 2 sheep:



With 3 sheep:


Seconds of gameplay (SOP) vs training time
Best SOP
SOP
Rolling avg SOP
Seconds of survival with all sheep alive, in a test game
Months of gameplay training (playing 24-7)
0
20
40
60
80
0
2
4
6
8
10
12

With 4 sheep:


Seconds of gameplay (SOP) vs training time
Best SOP
SOP
Rolling avg SOP
Seconds of survival with all sheep alive, in a test game
Months of gameplay training (playing 24-7)
0
10
20
30
40
50
0
2
4
6
8
10
12

All the test rounds in training have a lot of variance, this is probably due to the instability of the Q-table, only a little change and the helper will choose another path through the game.

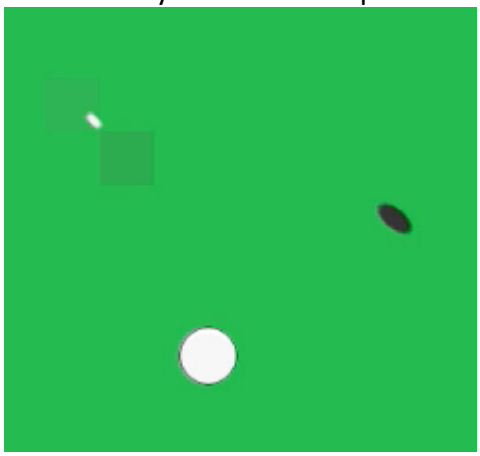
# of Sheep	Theoretical max number of states	Number of states reached	# of iterations	Seconds helper can avoid losing any sheep	Gameplay time it has trained
1	1 148	895	10 000 000	10 358	4 months
2	22 428	8 729	100 000 000	199	3 years
3	277 788	24 157	200 000 000	89	6 years
4	2 448 348	54 002	400 000 000	53	12 years

(Note here that for each iteration, we simulated 60 updates (60 FPS). Therefore, for training the 4 sheep with 400 000 000 iterations, the flocking code was run 400 000 000 * 60 (FPS) * 4(sheep) = 96 Billion times)

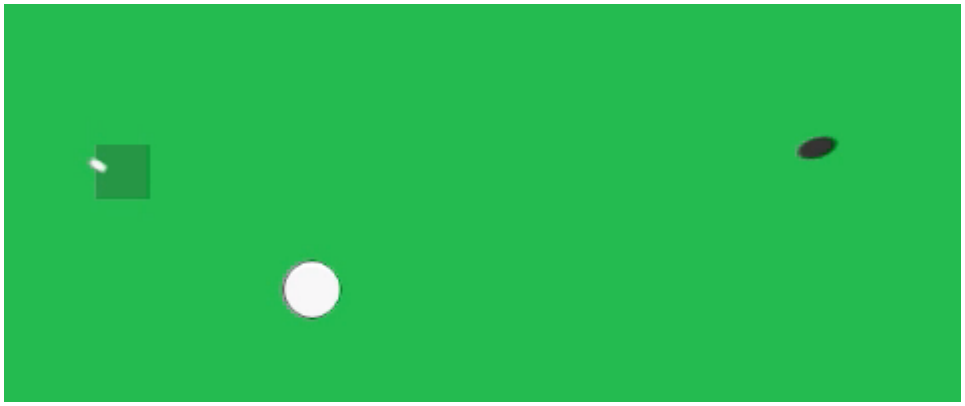
The helper trained on one 1 sheep, can play almost 3 hours without losing the sheep. This is because it learnt a loop, what it does is:

1. Push the sheep op towards the top left corner

2. Go a little away from the sheep to lure the wolf in



3. Get closer to the sheep to scare of the wolf



4. Go to step 1

[VIDEO this can be seen in the video: “2D Game - Q-lerning of helpers/1SheepLoop.mov”]
It learnt that getting the sheep too far away from the wolf, will make the wolf teleport closer to the sheep but at an unpredictable angle. Therefore, it goes away from the sheep to lure the wolf in. This loop takes about 25 seconds each round.
This loop is quite stable, but like any chaotic system the small differences for each round add up, and therefore it can “only” repeat this loop around 400 times.

The Q-table learnt for 1 sheep, can even recover from interference. If we “push” the sheep under the helper, the helper we get below the sheep, and start the loop again.
[VIDEO this can be seen in the video: “2D Game - Q-lerning of helpers/1SheepInterference.mp4”]

Also with 2 sheep, did the helper learn a loop, it goes as follows:

1. “push” sheep up and to the left, which will split them apart
2. Scare wolf away, which relying on the flocking to merge them again
3. Repeat

The same loop happened with 3 sheep [VIDEO this can be seen in the video: “2D Game - Q-lerning of helpers/3Sheep.mp4”]

4 Sheep is to many, it cannot move all sheep often enough and therefore starts to lose sheep to starvation.

All in all, the helper really learnt how to protect the sheep and how to keep them moving (so they do not starve to death). So, Q-learning for the 2D game have been a big success. Even more so when you consider that the helper is only supplementary to the farmer.
In the version of the 2D game we hand in, we have left the farmer to be controlled by the mouse, this is done purely so it is easier to play around with, and the Q-learning. In normal play the farmer is controlled by the arrow keys, but the game is quite easy with the helper, so we left the farmer controlled by the mouse to make it easier to play around with.

[1] Zhao, H.Y. 2014, "The Application and Research of C4.5 Algorithm", *Applied Mechanics and Materials,* vol. 513-517, pp. 1285-1288.

[2] “Tree algorithms: ID3, C4.5, C5.0 and CART”, https://medium.com/datadriveninvestor/tree-algorithms-id3-c4-5-c5-0-and-cart-413387342164

[3] “C4.5 Tutorial”, University of Regina, http://www2.cs.uregina.ca/~dbd/cs831/notes/ml/dtrees/c4.5/tutorial.html

[4] Byrisetty, "AGENT-BASED MODELING TO SIMULATE THE MOVEMENT OF A FLOCK OF BIRDS” https://pdfs.semanticscholar.org/bdb9/8e1ef68f78da2e0e0663032db9f7f4b39f22.pdf

[5] Melo, “Convergence of Q-learning: a simple proof”, http://users.isr.ist.utl.pt/~mtjspaan/readingGroup/ProofQlearning.pdf

[6] Marco Wiering, Martijn van Otterlo, "Reinforcement Learning in Continuous State and Action Spaces”, https://books.google.com/books?id=YPjNuvrJR0MC

4. Contributions

Haixiao Dai (s3678322)	Decision tree for fences in 3D game
Mads Bjørn (s3799147)	Q-learning for helper in 2D Game, and build the whole 2D and Q-learning
Yifan Wang (s3672150)	Deep Reinforcement Learning for Wolf in 3D game