

2021

## 304CR Game and AI CW2







GitHub: <https://github.coventry.ac.uk/wangh109/304CR-CW2-SoccerAI>  
Video:

HAN WANG

9987188

## Contents

Introduction .....	2
Rule-based AI .....	3
Attacker .....	4
Attacker .....	5
Pass Ball.....	6
 Attack Strategy .....	7
Defender .....	8
Ball in Sight.....	9
Defender.....	10
 patrol.....	11
Defence Strategy .....	12
Goalkeeper .....	12
Goal Patrol.....	13
 GamePlay .....	14
Machine Learning AI .....	17
ML-Agents .....	18
Training parameters.....	22
Self-play.....	22
Trainer types.....	23
Proximal Policy Optimisation (PPO) .....	24
MA-POCA (MultiAgent POsthumous Credit Assignment) .....	25
Whosyourdaddy .....	27
 Sprint .....	27
Gameplay .....	28
Conclusion .....	29
References .....	31

## Introduction

In this article, we have implemented a soccer AI game. The game is divided into three modes: Rule-based AI, Machine Learning AI, and Whosyourdaddy mode.

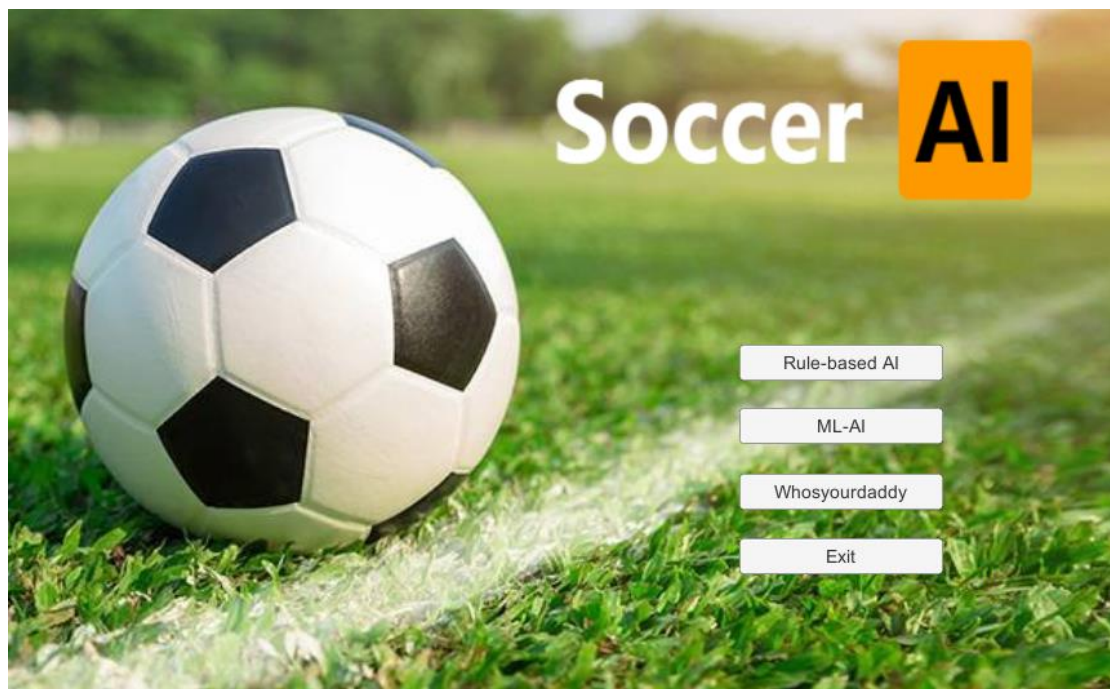
In Rule-based AI mode, we introduce two techniques commonly used in implementing AI (FSM and behaviour trees) and why we use behaviour trees rather than finite state machines. The behavioural scripts and logic of some of the key players in the game, such as attackers, defenders, goalkeepers etc., will be explained as well.

In Machine Learning AI, we introduce the APIs and training parameters in machine learning in Unity and how to optimise the parameters to maximise training results. We also compare the differences in training modes such as PPO and MA-POCA to achieve optimal teamwork training to solve the open problem of Team Cooperation in AI.

The Whosyourdaddy mode is a little egg in the game. The player gets epic enhancements to fight the AI and even acquired a new skill-sprint in order to beat the AI. Some of the gameplay improvement will also be explained in this section.



This element indicates that the author has considered gameplay in this section.



## Rule-based AI

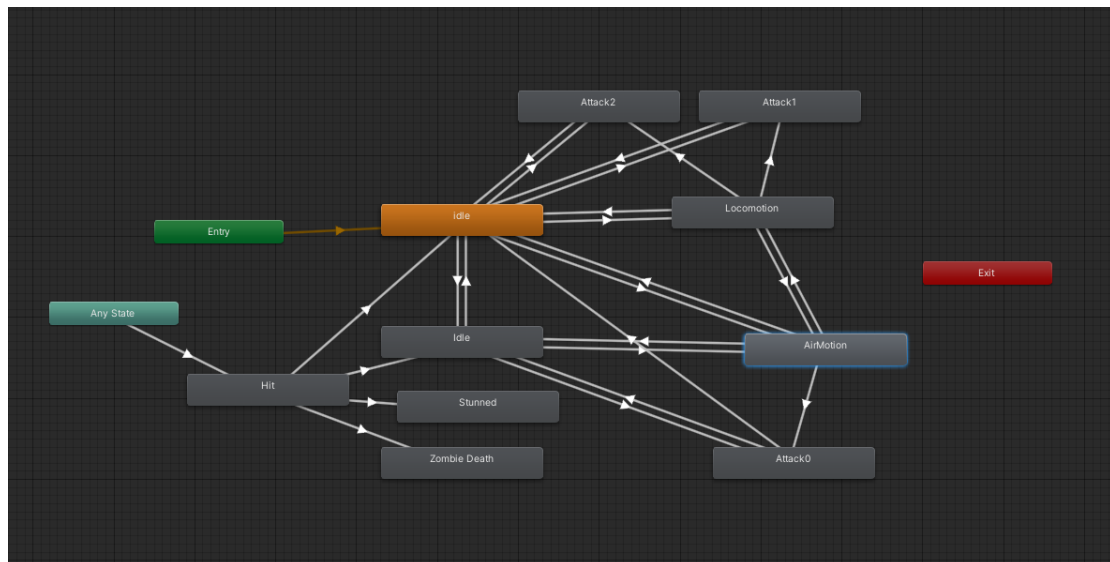
At present, it is common to use finite state machines(FSM) to design AI in games. Compared to hard-coded AI rules, FSM can reduce the code's complexity and create a more readable program by encouraging modular, reusable states. (Tsung, 2016)

An FSM stores all the states within a machine and describes the transition from one state to another. Finite State machines help define an exact number of states and events to trigger transitions. (Bors, 2018)

However, FSM also has some disadvantages. (Vorkflowengine, 2020)

- If the number of states is too large or uncertain, managing the states becomes challenging.
- When the algorithm is complex, the code becomes unreadable.
- If no valid input triggers the state, it may encounter an unreachable state and cause the behaviour to have an unknown outcome, leading to a system crash.

A good example is when we were working on the CW1 zombie shooter. We wanted to create a boss AI with many abilities, in other words, with many states, which was the beginning of the nightmare. The author spent plenty of time debugging the transitions between states to figure out why it is not switching to the desired state, which creates a lousy development experience.



Besides, it is pretty challenging to describe our soccer player behaviour states through FSM. For example, a soccer player has strategic behaviours, such as defending, attacking, and goalkeeping. These are not be solved by simply running, kicking, or Idle state. In this case, we would need to combine multiple

states, which would be another step into debugging hell.

FSMs are highly susceptible to human error, are difficult to modify as transitions are added. As a result, FSM are limit in designing complex AI systems, and this is where behaviour trees step in. (Allerin, 2020)

A behaviour tree consists of interconnected behaviours of nodes in the form of a tree, starting from a single root node. Behaviour trees can help create an efficient way to design complex algorithms and modular and reactive systems. It removes the direct jump logic of finite state machine states as it turns states into behaviours. The type of the parent node determines the jumping of different behaviours. (Colledanchise, 2017)

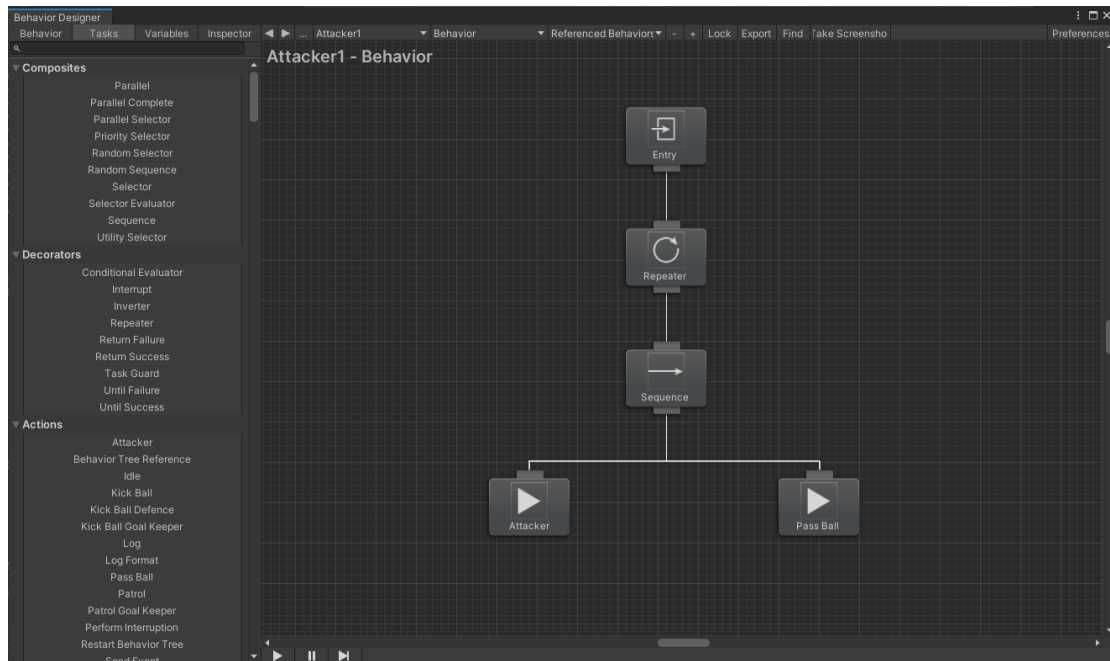
The significant difference between behaviour trees and state machines is that behaviour trees trigger individual behaviours through conditions. Whatever conditions are met, the behaviour is executed. The advantage of this is that each behaviour can be extracted separately and configured individually. The different behaviours can be customised later by simply adding the corresponding conditions. Most importantly, it can prevent us from going to debugging hell.

In summary, a behaviour tree suits our needs best other than FSM. We will use the behaviour tree Combined with a Rule-based AI approach to complete this soccer game. To be specific, the behaviour tree structure will be provided by Behavior Designer to visualise the script and help faster the game development progress. We will focus on implementing the action node scripts below it.

## **Attacker**

This behaviour tree for the Attacker consists of two action nodes, a repeater node and a sequence node. The repeater node repeats its child task's execution, and the sequence node is similar to an "and" operation. It will only run the next task in turn if the first subtask returns a success.

The Attacker's behaviour is to repeatedly execute the attacker script and find any chance to passing the ball to the player near the goal.



## Attacker

The On-Update function of the attacker script first gets the distance between the agent itself and the ball. If the ball is within kickable range, it executes the kick.

Suppose the ball is not within kickable range. In that case, we need to dynamically assign agent position based on how close the agent is to the ball. The implementation of this part is explained in the Attack Strategy.

```
Assembly-CSharp soccerAI.Attacker
44 }
45
46 99+ 个引用
47 public override TaskStatus OnUpdate()
48 {
49     //Get the position of the ball:
50     ballLocation = agent.getBallPosition();
51     //Get the position of itself:
52     agentLocation = agent.transform.position;
53     //If the football is within kickable range:
54     if (Condition.CanKickBall(agentLocation, ballLocation))
55     {
56         //Towards the ball:
57         agent.transform.LookAt(ballLocation);
58         //Depending on the team, give the ball a force:
59         if (isLeft)
60         {
61             Ball.AddForce(ballLocation, Define.RightDoorPosition);
62         }
63         else
64         {
65             Ball.AddForce(ballLocation, Define.LeftDoorPosition);
66         }
67         //return success;
68         return TaskStatus.Success;
69     }
70     else
71     {
72         //Get agent position inside the attacking formation:
73         targetLocation = AttackStrategy.Instance.GetAttackGroupLocation(agent, ballLocation, isLeft);
74         //Set the target points:
75         agent.setDestination(targetLocation);
76         //return running;
77         return TaskStatus.Running;
78     }
79 }
80
81 }
```

## Pass Ball

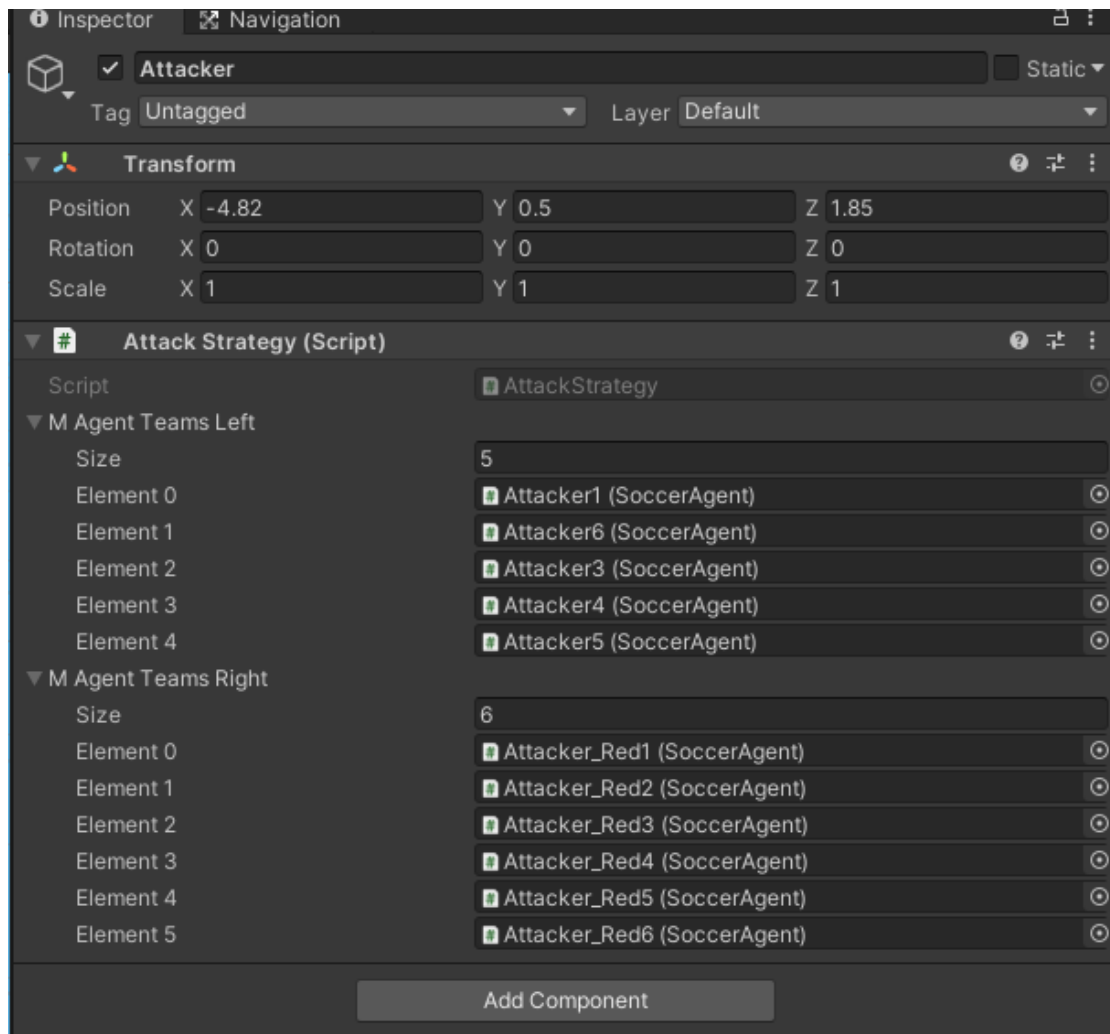
In this script, we determine the nearest player by calling the findNear method in the Attack Strategy. If we have the ball and are not the nearest player to the goal, we pass the ball to the nearest player to complete the goal action.

```
99+ 个引用
public override TaskStatus OnUpdate()
{
    nearPlayer = AttackStrategy.Instance.findNear(mAgent, bLeft);
    if(nearPlayer.getNum() == mAgent.getNum())
    {
        //if we are the closest player to the goal return failure
        return TaskStatus.Failure;
    }
    else
    {
        // if we are not the closest player to the goal.
        if (Condition.CanKickBall(mAgent.transform.position, ballLocation))
        {
            mAgent.transform.LookAt(ballLocation);
            //shoot the ball to the nearest attacker
            ball.AddForce(ballLocation, nearPlayer.transform.position);
            return TaskStatus.Success;
        }
        return TaskStatus.Failure;
    }
}
}
```



## Attack Strategy

The attacking AI is a group strategy. Each player is dynamically assigned to a role according to the ball's distance, thus deciding whether to execute the kicking AI or the following AI. It can be seen as a simple flocking behaviour as well. (Fathy et al., 2014)



The GetAttackGroupLocation function receives the agent, target position, and the team as the parameters. It returns a Vector3 parameter to determine the agent's position in the flocking group



```

    /// <summary>
    /// Get the position in the attacker team.
    /// </summary>
    /// <param name="agent"></param>
    /// <param name="targetPosition"></param>
    /// <param name="isLeft"></param>
    /// <returns></returns>
    1 个引用
    public Vector3 GetAttackGroupLocation(SoccerAgent agent, Vector3 targetPosition, bool isLeft)
    {
        //Get a list of players according to the team
        List<SoccerAgent> team = GetAgentTeam (isLeft);

        //Rank the distance between the player and the target point, with the closest to the target point will be set at the top of the list.
        team.Sort((a, b) => {
            return Vector3.Distance(a.transform.position, targetPosition).CompareTo(Vector3.Distance(b.transform.position, targetPosition));
        });

        //Get the index value after sorting;
        var index = team.FindIndex (a=>a.getNum()==agent.getNum());

        // Index value of 0 is the closest player to the target point;
        if (index == 0)
        {
            return targetPosition;
        }
        else
        {
            // Get the position of the closest player to the target point;
            var nearstBallAgentLocation = team[0].transform.position;
            int position = (index + 1) / 2;

            if ((index%2)==0)
            {
                //If the index is even, place it on the side of the player closest to the target point;
                return new Vector3(nearstBallAgentLocation.x, 0, nearstBallAgentLocation.z - position * 6);
            }

            //If the index is an odd number, place the side of the player nearest to the target point, above the pitch;
            return new Vector3(nearstBallAgentLocation.x, 0, nearstBallAgentLocation.z + position * 6);
        }
    }
}

```

To make three agents move together as an attacking group towards the soccer. The agents closest to the ball kicks the ball. The other two agents move to either side of the closest agent to protect and cover the attacking agent. The flocking behaviour is achieved by sorting the players according to their proximity to the ball. The target point of the player closest to the ball is the given target point. Agents who are further away from the ball have added a distance back according to their number to achieve the following effect.

Find Near function sorts the agents according to their distance from the goal, and the agent with index value 0 will be returned.

```

    ///Find the player closest to the goal to the pass ball
    1 个引用
    public SoccerAgent findNear(SoccerAgent agent, bool isLeft)
    {
        List < SoccerAgent > team = GetAgentTeam(isLeft);

        if (isLeft)
        {
            team.Sort((a, b) =>
            {
                return Vector3.Distance(a.transform.position, Define.RightDoorPosition).CompareTo(Vector3.Distance(b.transform.position, Define.RightDoorPosition));
            });
        }
        else
        {
            team.Sort((a, b) =>
            {
                return Vector3.Distance(a.transform.position, Define.LeftDoorPosition).CompareTo(Vector3.Distance(b.transform.position, Define.LeftDoorPosition));
            });
        }

        return team[0];
    }
}

```

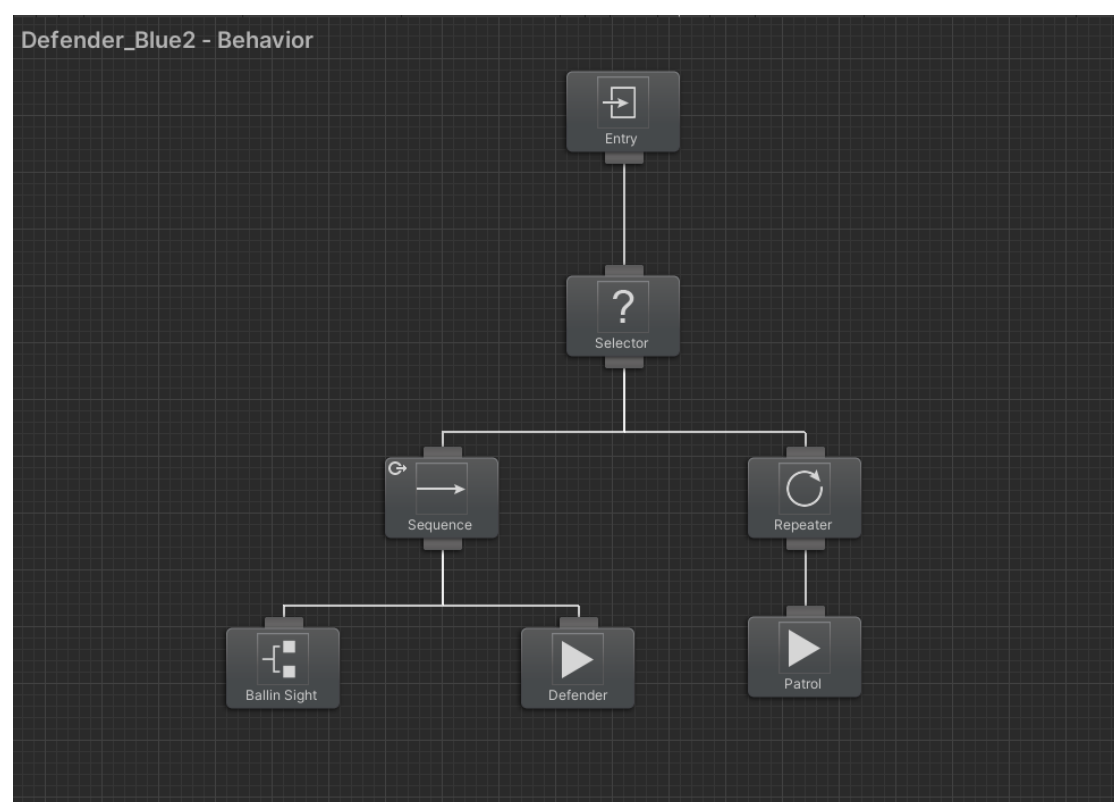
## Defender

In the Defender behaviour tree, we have added a new node which is a selector node. A selector node is similar to an "or" operation. It returns success as one

of its child tasks returns success.

The defensive strategy is to execute the patrol AI if the ball is not in the agent's sight. Once the agent sees the ball, the Ball-in-Sight script return success, then it will execute the Defender script under the sequence node to defend. Once the ball is out of the field of view, the agent will continue to patrol.

Note that the sequence node means that the right-hand logic will be executed first if the player cannot see the ball. As soon as the player can see the ball, the patrolling behaviour will be interrupted. The defender action node will be executed.



## Ball in Sight

Ball in Sight script is used to determine if the player can see the ball. This is conditional logic as it needs to inherit from the Behavior Designer's Condition node. It returns true when the ball is close to the agent. Otherwise, it returns failure.

```

public class BallinSight : Conditional
{
    /// <summary>
    /// agnet;
    /// </summary>
    private SoccerAgent Agent;
    /// <summary>
    /// ball;
    /// </summary>
    private Ball Ball;

    99+ 个引用
    public override void OnStart()
    {
        Agent = GetComponent<SoccerAgent>();
        Ball = Agent.getBall().GetComponent<Ball>();
    }

    99+ 个引用
    public override TaskStatus OnUpdate()
    {
        //If can see the ball, return success. Otherwise return failure;

        if (Condition.CanSeeBall(Agent.transform.position, Ball.transform.position))
        {
            return TaskStatus.Success;
        }
        else
        {
            return TaskStatus.Failure;
        }
    }
}

```

## Defender

The Defender script is similar to the Attacker so that we will not go into too much detail here.

```

23 public override TaskStatus OnUpdate()
24 {
25     //Get football position;
26     ballLocation = agent.getBallPosition();
27     //Get agent position;
28     agentLocation = agent.transform.position;
29     //To get into this node the agent must have seen the ball
30     if (Condition.CanKickDefence(agentLocation, ballLocation))
31     {
32         //if close to the ball
33         if (Condition.CanKickBall(agentLocation, ballLocation))
34         {
35             //face to the ball ;
36             agent.transform.LookAt(ballLocation);
37             //Depending on the direction of opponent's goal, give the ball a force;
38             bool isLeft = agent.WhichTeam();
39             if (isLeft)
40             {
41                 Ball.AddForceBig(ballLocation, Define.RightDoorPosition);
42             }
43             else
44             {
45                 Ball.AddForceBig(ballLocation, Define.LeftDoorPosition);
46             }
47             //Return successful;
48             return TaskStatus.Success;
49         }
50         else
51         {
52             if (Condition.CanDefenceGroup(agent.WhichTeam(), agent.transform.position, ballLocation)) {
53                 var target = DefenceStrategy.Instance.GetDefenceGroupLocation(agent, ballLocation, isLeft);
54                 agent.setDestination(target);
55                 return TaskStatus.Running;
56             }
57             else {
58                 //agent can defence the ball, but the distance is not enough to kick the ball, so move towards the ball.
59                 agent.setDestination(ballLocation);
60             }
61             return TaskStatus.Running;
62         }
63     }
64     // return Failure when can't defence the ball;
65     return TaskStatus.Failure;
66 }
67

```



Patrolling is a common AI strategy in the game based on the principle of moving up, down, left, and right at the initial position. (Schwab, 2009)

In CW1, we have implemented a random patrol zombie AI to use the Random method to create a random value and add them into the zombie destination vector3 to achieve random movement.

```
anim.SetFloat("Speed", Speed);
//choose a random direction to move
Vector3 randomRange = new Vector3((Random.value - 0.5f) * 2 * 15.0f, 0, (Random.value - 0.5f) * 2 * 15.0f);
Vector3 nextDestination = transform.position + randomRange;
agent.destination = nextDestination;
```

However, this approach causes discontinuous patrol points to be selected, which causes agents to scurry around like headless flies, running to the top left corner and then to the top right corner, giving people a haphazard feeling.

In this project, we use an alternative approach to improve this problem: to number each patrol point and then move them in order. This method will have a certain coherence. We use this approach because we do not want agents to run around the field meaninglessly in a soccer game.

```
12 个引用
public override void OnAwake()
{
    agent = GetComponent<NavMeshAgent>();

    var InitPos = agent.transform.position;
    var PatrolPos1 = new Vector3(InitPos.x + Patrol_Circle, InitPos.y, InitPos.z + Patrol_Circle);
    var PatrolPos2 = new Vector3(InitPos.x + Patrol_Circle, InitPos.y, InitPos.z - Patrol_Circle);
    var PatrolPos3 = new Vector3(InitPos.x - Patrol_Circle, InitPos.y, InitPos.z - Patrol_Circle);
    var PatrolPos4 = new Vector3(InitPos.x - Patrol_Circle, InitPos.y, InitPos.z + Patrol_Circle);

    waypoints.Add(PatrolPos1);
    waypoints.Add(PatrolPos2);
    waypoints.Add(PatrolPos3);
    waypoints.Add(PatrolPos4);
}

// Update is called once per frame
99+ 个引用
public override TaskStatus OnUpdate()
{
    agentPos = agent.transform.position;
    // When reach the patrol point, re-randomized to the next patrol point
    if (Mathf.Abs(agentPos.x - PatrolPos.x) < 1 && Mathf.Abs(agentPos.z - PatrolPos.z) < 1)
    {
        index = (index + 1) % waypoints.Count;
        PatrolPos = waypoints[index];
        //Debug.Log("PatrolPos: " + PatrolPos);
    }
    // Set the patrol point as the target point of Agent;
    agent.SetDestination(PatrolPos);
    return TaskStatus.Running;
}
```

## Defence Strategy

The defence strategy is Similar to the attack strategy, except that the formation becomes aligned in a straight line.

```
public Vector3 GetDefenceGroupLocation(SoccerAgent agent, Vector3 targetPosition, bool isLeft)
{
    //Get list of agent according the team;
    List<SoccerAgent> team = GetAgentTeam(isLeft);

    //Rank the distance between the player and the target point, the closest to the target point the top of the list;
    team.Sort((a, b) => {
        return Vector3.Distance(a.transform.position, targetPosition).CompareTo(Vector3.Distance(b.transform.position, targetPosition));
    });

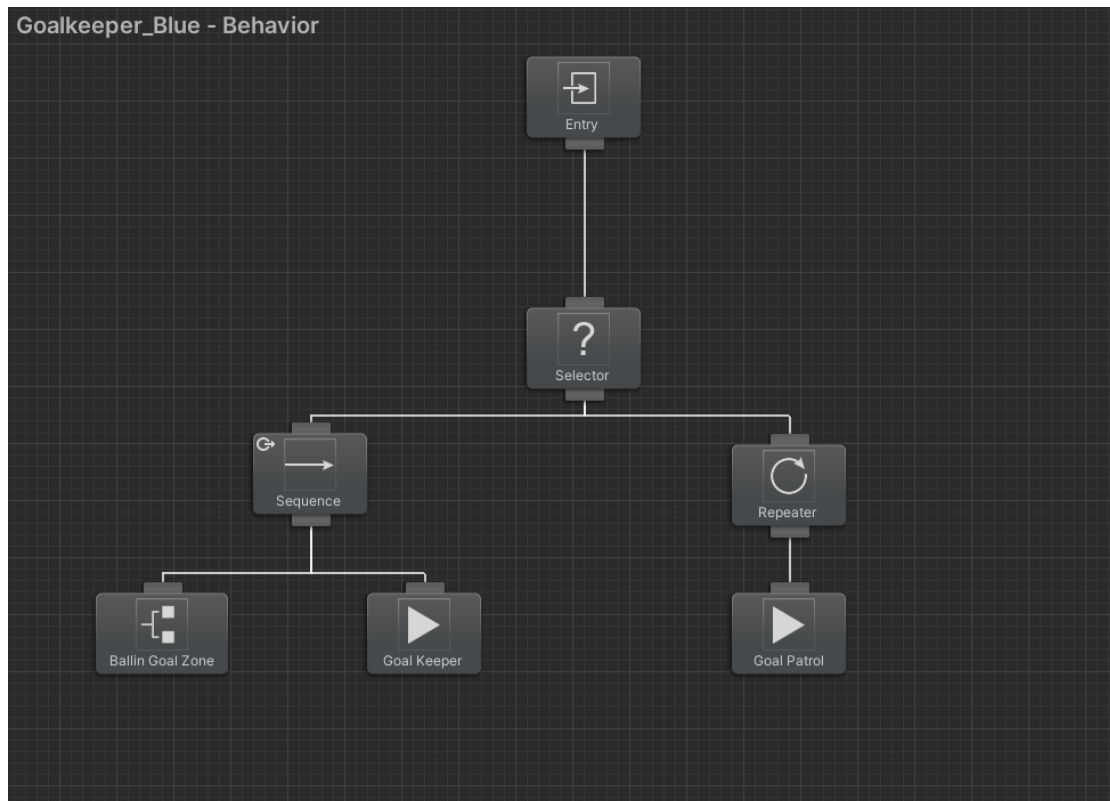
    //Get agent index value after sorting.
    var index = team.FindIndex(a => a.getNum() == agent.getNum());

    // if the index value is 0, the target is the ball.
    if (index <= 0)
    {
        return targetPosition;
    }
    else
    {
        //Align the agent to the nearest player who is closer to the ball
        var nearsMeAgentLocation = team[index-1].transform.position;

        if (transform.position.z > nearsMeAgentLocation.z)
        {
            return new Vector3(nearsMeAgentLocation.x, 0, nearsMeAgentLocation.z + 3);
        }
        else
        {
            return new Vector3(nearsMeAgentLocation.x, 0, nearsMeAgentLocation.z - 3);
        }
    }
}
```

## Goalkeeper

After introducing the attacking AI and defending AI, we will discuss the most critical role on the pitch - the goalkeeper. The goalkeeper's AI is very similar to the defensive AI, but the goalkeeper generally only needs two patrol points, i.e., on either side of the goal.



## Goal Patrol

In the goalkeeper's patrol logic, we only need to patrol along the sides of the goal at fixed points.

```

public override void OnStart()
{
    agent = GetComponent<SoccerAgent>();
    ballLocation = agent.getBall().transform;
    //Get the location of the Agent itself
    Vector3 InitPos = agent.transform.position;
    //Set up patrol waypoints
    PatrolPositions.Add(new Vector3(InitPos.x, InitPos.y, InitPos.z + Define.Patrol_Circle));
    PatrolPositions.Add(new Vector3(InitPos.x, InitPos.y, InitPos.z - Define.Patrol_Circle));

    //Choose a patrol point near to the local position
    float distance = Mathf.Infinity;
    //Distance difference between local position and the patrol point
    float localDistance;
    for(int i = 0; i < PatrolPositions.Count; ++i)
    {
        if((localDistance = Vector3.Magnitude(agent.transform.position - PatrolPositions[i])) < distance)
        {
            distance = localDistance;
            index = i;
        }
    }

    PatrolPos = PatrolPositions[index];
    agent.setDestination(PatrolPos);
}

99+ 个引用
public override TaskStatus OnUpdate()
{
    //If a player moves to a patrol point, set the next patrol point
    agentPosition = agent.transform.position;
    if(Mathf.Abs(agentPosition.x - PatrolPos.x) < 1 && Mathf.Abs(agentPosition.z - PatrolPos.z) < 1)
    {
        index = (index + 1) % PatrolPositions.Count;
        PatrolPos = PatrolPositions[index];
    }

    //Moving players to patrol points
    agent.setDestination(PatrolPos);

    agent.transform.LookAt(ballLocation);
    return TaskStatus.Running;
}

```



## GamePlay

In Rule-based AI mode, we use a Define script and Condition script to define the game's reusable variables and conditions. These values are iteratively tuned to achieve the best possible gameplay.

For example, Force is the amount of force applied by the agent when kicking the ball. This value is relative to the length of the pitch to ensure that the ball is not too quickly kicked into the goal or force too small to be insignificant.

```

namespace soccerAI
{
    25 个引用
    public class Define
    {
        /// <summary>
        /// Football playground Length
        /// </summary>
        public static int Length = 8;
        /// <summary>
        /// The force of the kick
        /// </summary>
        public static int FORCE = 10;
        /// <summary>
        /// The big force of kick
        /// </summary>
        public static int BIG_FORCE = 20;
        /// <summary>
        /// Radius of patrol
        /// </summary>
        public static float Patrol_Circle = 1.5f;
        /// <summary>
        /// see the radius of the ball
        /// </summary>
        public static float See_Circle = 10f;
        /// <summary>
        /// Location of the left-hand goal of the pitch
        /// </summary>
        public static Vector3 LeftDoorPosition = new Vector3(-Length/2f, 0, 0);
        /// <summary>
        /// Position of the right-hand goal of the pitch
        /// </summary>
        public static Vector3 RightDoorPosition = new Vector3(Length/2f, 0, 0);
        /// <summary>
        /// Distance to be able to kick the ball
        /// </summary>
        public static float CanKickBallDistance = 1f;
    }
}

```

未找到相关问题

The CanSeeBall in Condition will take two Vector 3 parameters, the agent's position and the position of the ball. These two parameters are used to determine whether the player can see the ball or not based on the See\_Circle in Define.



```

public class Condition
{
    2 个引用
    public static bool CanSeeBall(Vector3 agentLocation, Vector3 ballLocation)
    {
        if (Mathf.Abs(agentLocation.x - ballLocation.x) < Define.See_Circle && Mathf.Abs(agentLocation.z - ballLocation.z) < Define.See_Circle)
        {
            if (ballLocation.y <= 0.5f)
            {
                return true;
            }
        }
        return false;
    }

    0 个引用
    public static bool CanSeeDoor(Vector3 agentLocation, bool isLeft)
    {
        if (isLeft)
        {
            if ((Define.RightDoorPosition - agentLocation).sqrMagnitude < 400)
            {
                return true;
            }
            return false;
        }
        else
        {
            if ((Define.LeftDoorPosition - agentLocation).sqrMagnitude < 400)
            {
                return true;
            }
            return false;
        }
    }

    5 个引用
    public static bool CanKickBall(Vector3 agentLocation, Vector3 ballLocation)
    {
        if (ballLocation.y >= 2)
        {
            return false;
        }
        if (Mathf.Abs (agentLocation.x - ballLocation.x) < Define.CanKickBallDistance
            && Mathf.Abs (agentLocation.z - ballLocation.z) < Define.CanKickBallDistance)
            return true;
        return false;
    }
}

```

There are many more Define arguments and Condition functions that can be looked up in the Github code and what they mean, so I won't go into them here.

## Machine Learning AI



In the rule-based AI scenario, we have pre-code the agent's behaviour. However, this is not enough. Once the player is familiar with the agent's behaviour, the agent will become too easy to beat. We need another way to make the player unable to predict the AI's actions to increase the gameplay. Some of the more complex AI systems use genetic algorithms or machine learning to design algorithms that produce different behaviours without pre-coding them.

Genetic algorithm is a search technique inspired by Darwin's theory of natural evolution. It naturally reflects the selection process of the most suitable elements. It is often used for search-based optimisation problems that are difficult and time-consuming to solve with other general-purpose algorithms. A genetic algorithm can perform a directed search of the solution space. (Hamed, 2010) For example, Watson et al. (2008) have implemented a genetic algorithm to optimise city development for a turn-based strategy game-Freeciv. The result shows using a GA in the Freeciv can significantly reduce the need for micro-managing various facets of city development.

Machine learning (ML) uses neural network models to simulate the human brain to solve problems that conventional algorithms cannot solve, such as object recognition or natural language processing. Neural network models are non-linear tools for modelling statistical data. They could find patterns in any input and output data.

On the other hand, machine learning, or neural networks, usually work on continuous data such as board games. (Baeldung, 2020) L. Samuel et al. (2000) demonstrated that machine learning could learn to play checkers better than people who write programs in a short learning time (8 to 10 hours).

Of course, there are thousands of variants of both, so the line between GA and ML is somewhat blurred. Sehgal et al. (2019) propose a parameter value that uses a genetic algorithm to find depth-deterministic policy gradients combined with post hoc empirical replay to help speed up the machine learning process. The research shows that combining the two techniques produces good results in improving training speed.

In this section, we have chosen to use Machine Learning because machine learning has gained considerable popularity in recent years. Besides, there is an MLAgent library available in Unity, which can significantly facilitate the

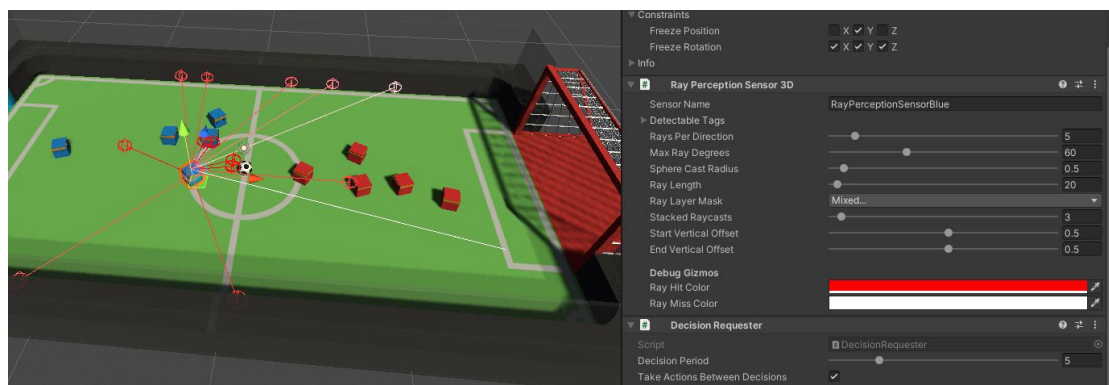
development process.

## ML-Agents

The Unity Machine Learning Agents Toolkit (ML-Agents Toolkit) is a tool to training agents for machine learning through the use of the Unity game environment by using CUDA and PyTorch. (Simonini, 2020)

In this scenario, we will use the MLAgents to complete the training of AI. We decided only to train one AI because no matter which team is agents belong to, their mindset is the same.

MLAgents fully control soccer players' actions through the Ray Perception Sensor component. The player can submit information about the game environment to MLAgents, which calculates an array of actions to move the player.



The MyAI script implements the following five methods:

- Initialise method to set some properties of the player, such as which team the player belongs to. The Initialisation of the Team variable based on the Team ID under the Behavior Parameters component.

```

public override void Initialize()
{
    behaviorParameters = gameObject.GetComponent<BehaviorParameters>();

    if (behaviorParameters.TeamId == (int)Team.Red)
    {
        team = Team.Red;
        iniPos = new Vector3(transform.position.x - 4f, 1f, transform.position.z);
    }
    else {
        team = Team.Blue;
        iniPos = new Vector3(transform.position.x + 4f, 1f, transform.position.z);
    }

    HorizontalSpeed = 0.4f;
    VerticalSpeed = 1.1f;
    rb = GetComponent<Rigidbody>();
    rb.maxAngularVelocity = 500;
    environmentParameters = Academy.Instance.EnvironmentParameters;
}

```

- The OnActionReceived method used to move agents and receives an array of operations returned by MLAgents. The array has three values, each of which is of type Int and can only take 0, 1, and 2.

For example, if the zero position of the array is 1 the agent moves forward, 2 is backward. If the first position of the array is 1, the agent moves right, and 2 moves left. If the second position of the array is 1, the agent moves left; if it is 2, the agent moves right.

```

public override void OnActionReceived(ActionBuffers actionBuffers)
{
    var moveDirection = Vector3.zero;
    var rotateDirection = Vector3.zero;
    kickPower = 0f;
    var act = actionBuffers.DiscreteActions;
    var forward = act[0];
    var right = act[1];
    var rotate = act[2];

    switch (forward)
    {
        case 1:
            moveDirection = transform.forward * VerticalSpeed;
            kickPower = 1f;
            break;
        case 2:
            moveDirection = transform.forward * -VerticalSpeed;
            break;
    }
    switch (right)
    {
        case 1:
            moveDirection = transform.right * HorizontalSpeed;
            break;
        case 2:
            moveDirection = transform.right * -HorizontalSpeed;
            break;
    }
    switch (rotate)
    {
        case 1:
            rotateDirection = transform.up * -1f;
            break;
        case 2:
            rotateDirection = transform.up * 1f;
            break;
    }
    transform.Rotate(rotateDirection, Time.deltaTime * 100f);
    AddReward(-m_Existential);
    //2f is the agent run speed
    rb.AddForce(moveDirection * 2f, ForceMode.VelocityChange);
}

```

- The heuristic will generate a heuristic corresponding to the values of the "Horizontal" and "Vertical" input axes (corresponding to the keyboard arrow keys) Action. (This part of the code is taken from the official MAgent example).

```

14 个引用
public override void Heuristic(in ActionBuffers actionsOut)
{
    var discreteout = actionsOut.DiscreteActions;
    discreteout.Clear();

    //forward
    if (Input.GetKey(KeyCode.W))
    {
        discreteout[0] = 1;
    }
    if (Input.GetKey(KeyCode.S))
    {
        discreteout[0] = 2;
    }

    //rotate
    if (Input.GetKey(KeyCode.A))
    {
        discreteout[2] = 1;
    }
    if (Input.GetKey(KeyCode.D))
    {
        discreteout[2] = 2;
    }

    //right
    if (Input.GetKey(KeyCode.E))
    {
        discreteout[1] = 1;
    }
    if (Input.GetKey(KeyCode.Q))
    {
        discreteout[1] = 2;
    }
}

```

- The OnCollisionEnter is the collision event handling function. When the agent moves forward and hits the ball, we give the ball a force. If the colliding body is a ball, get the position of the impact point, subtract the position of the agent, and take this direction as the forward direction of the ball.

```

// ball kick function
@Unity 消息 | 0 个引用
private void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.CompareTag("ball"))
    {
        // Vector3 direction = collision.gameObject.transform.localPosition - this.transform.localPosition;

        AddReward(.2f * m_BallTouch);
        Vector3 direction = collision.contacts[0].point - this.transform.localPosition;
        direction = direction.normalized;
        float force = 2000.0f * kickPower;
        collision.rigidbody.AddForce(direction * force);
    }
}

```

## Training parameters

This is the parameter file we used for training. We will focus on some of the parameters in the following chapter.

```
behaviors:
  Soccer:
    trainer_type: poca
    hyperparameters:
      batch_size: 2048
      buffer_size: 20480
      learning_rate: 0.0003
      beta: 0.005
      epsilon: 0.2
      lambda: 0.95
      num_epoch: 3
      learning_rate_schedule: constant
    network_settings:
      normalize: false
      hidden_units: 512
      num_layers: 2
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    keep_checkpoints: 5
    max_steps: 5000000
    time_horizon: 1000
    summary_freq: 10000
    threaded: false
    self_play:
      save_steps: 50000
      team_change: 200000
      swap_steps: 2000
      window: 10
      play_against_latest_model_ratio: 0.5
      initial_elo: 1200.0
```

## Self-play

Andrew Cohen (2020) mentions the self-play feature in ML-agent, which means no human intervention is needed to get data input which is exactly what we need for our game, as we do not have as much time to collect data from human players, so training two teams of AI to play against each other is the optimal choice.

Through self-play, agents learn by competing against past versions of their opponents in an adversarial game

```
27     self_play:
28         save_steps: 50000
29         team_change: 250000
30         swap_steps: 2000
31         window: 10
32         play_against_latest_model_ratio: 0.5
33         initial_elo: 1200.0
34
```

We set the value of `save_steps` to 50000 because an enormous `save_steps` value will produce a group of opponents covering a more comprehensive range of skill levels and possible playing styles as the policy receives more training.

The official recommendation for `team_change` is  $5 * \text{save\_steps}$ . In our case, this value is 250000. The greater the value is, the longer the agent has to train against his opponents. The longer agent trains against the same set of opponents, the better they will defeat them. However, training them for too long may lead to over-adaptation to a particular opponent's strategy so that the agent may fail in training against the next group of opponents.

A higher value of `play_against_latest_model_ratio` indicates that the agent will play more often against the current opponent. As the agent is updating its strategy, the opponent will be different for each iteration. This may lead to an unstable learning environment but presents the agent with an automatic course of more challenging situations, leading to a more assertive final policy. To balance the two, we set the value to 0.5

The self-gaming framework can also create some practical problems. Specifically, the non-stationarity of the transition function (i.e. changing opponents) may lead to over-fitting to beat a particular play and instability in the training process.

There are many more parameters which, due to word limits, we will then move on to the selection of trainer types.

## Trainer types

There are a variety of common training approaches, both Proximal Policy Optimisation (PPO) and Soft Actor-Critic (SAC). in MAgent can be used for training using self-play.

However, from our football game, individual football agents possess unstable dynamics which can lead to significant problems with the empirical replay



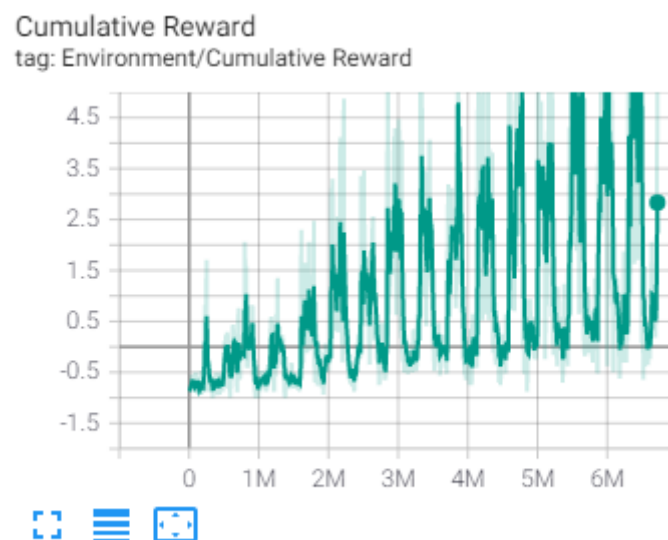
mechanism used by SAC. Therefore, we use a PPO approach to train our agents.(Foerster et al., 2017)

### Proximal Policy Optimisation (PPO)

PPO is a stable, flexible DRL algorithm developed by Schulman et al. (2017) as a default algorithm in OpenAI and ML-agents.

The PPO algorithm is a policy-based algorithm that collects a number of samples and learns how to improve itself by increasing the likelihood of taking incentives and decreasing the likelihood of not taking incentives by using a sample of current policies.

At the beginning of the project, we used the PPO training method, and the training results showed good results that the agents are cumulative get reward.



After more than six million steps of training, the agents have been able to score quickly in a concise period; however, the actual game result is a big drop. Agents shot from very tricky angles and often scored on the kick-off. This was because there were no agents in front of the goal to block the ball. This is not what we want to see in a football match, as there is not reflecting the true essence of soccer as a competitive team game.

## MA-POCA (MultiAgent POsthumous Credit Assignment)



Unity introduced the Mo-PoCA- training method in the recent March ML agent update, which could effectively solve the problem of multi-agent cooperation.

There have been a number of open problems in the AI field, and cooperation between multiple agents is formally one of them.

This category of research is collectively referred to as collaborative AI. Due to the interaction between agents, the complexity of multi-agent problems increases rapidly with the complexity of the agents' behaviour or the number of agents and the complexity of the environment, and how to solve the cooperation as well as the interaction between multiple agents becomes one of the development challenges in this field. (Panait & Luke, 2005)

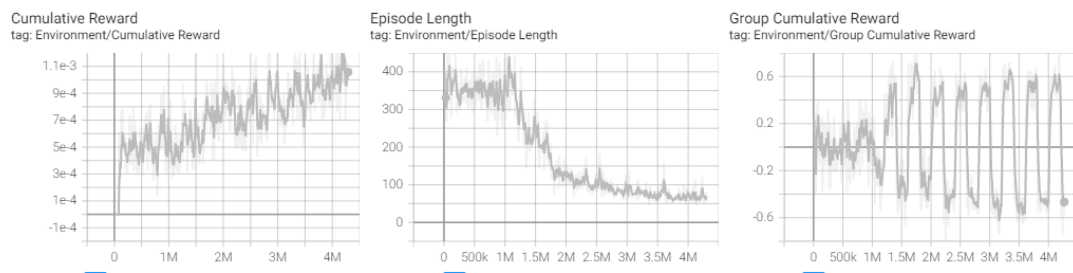
A multi-agent system (MAS) is a computer system composed of multiple interacting intelligent agents (J. Hu et al., 2020). We can use POCA to simulate a training MAS environment. A POCA training cooperative behaviours links groups of agents working towards a common goal, where the individual's success is linked to the whole group's success.

In POCA, instead of an individual agent get rewarded, the whole team is rewarded. Each agent in the team will be rewarded in varying amounts depending on their actions.

```
2 个引用
public void GoalTouched(Team scoredTeam)
{
    if (scoredTeam == Team.Blue)
    {
        TeamBlue.AddGroupReward(1 - m_ResetTimer / MaxEnvironmentSteps);
        TeamRed.AddGroupReward(-1);
    }
    else
    {
        m_TeamRed.AddGroupReward(1 - m_ResetTimer / MaxEnvironmentSteps);
        TeamRed.AddGroupReward(-1);
    }
    TeamBlue.EndGroupEpisode();
    TeamRed.EndGroupEpisode();
    ResetScene();
}
```

After the training, the results show that agents are steadily learning how to attack and defend as a team, and the time spent per game is decreasing accordingly. The cumulative team reward also tends to be a normally distributed value which is a good thing.

## Environment



In particular, the game shows that the agent is able to play offensively and defensively with awareness, and, of course, some physical confrontation appears in the game as well.

## Whosyourdaddy

Whosyourdaddy is a cheat code in Warcraft 3 means god mode. We will deify the player in this mode: the human player will increase in size to twice the size and a more powerful sprint function. This is to reduce the frustration of being defeated by AI.



### Sprint

The player can press the space bar to sprint in the forward direction. The duration and intensity of the sprint have been considered to achieve the best possible gameplay.

```

1 个引用
void Sprint(float h, float v)
{
    transform.Translate((Vector3.forward * -h + Vector3.right * v) * speed * Time.deltaTime);

    if (!isSprint)
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            isSprint = true;

            directionX0Z = transform.forward*-h + transform.right * v;

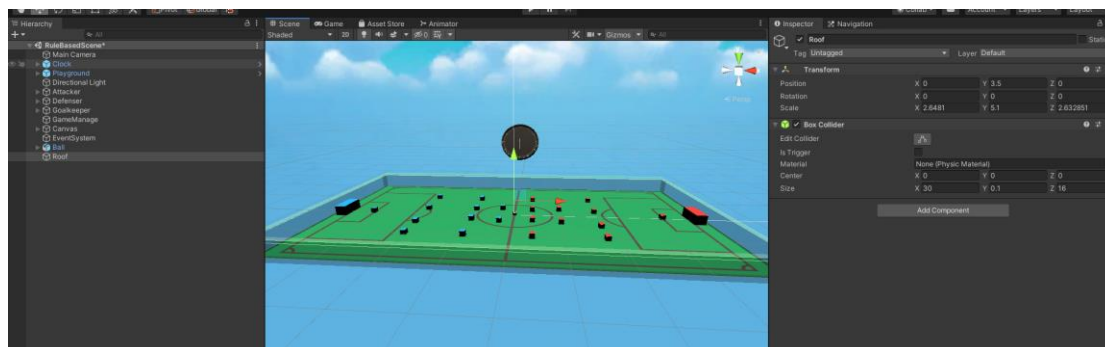
            directionX0Z.y = 0f;
        }
    }
    else
    {
        if (SprintTime <= 0) // reset
        {
            isSprint = false;

            SprintTime = sprintDuration;
        }
        else
        {
            SprintTime -= Time.deltaTime;
            rb.velocity = directionX0Z * SprintTime * sprintSpeed;
        }
    }
}

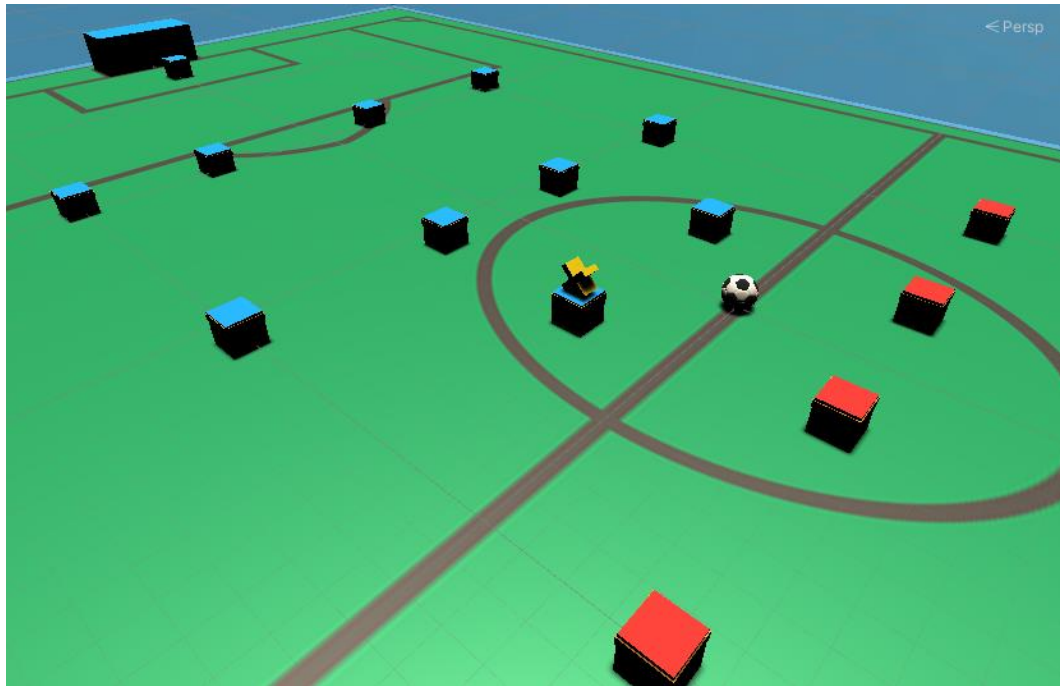
```

## Gameplay

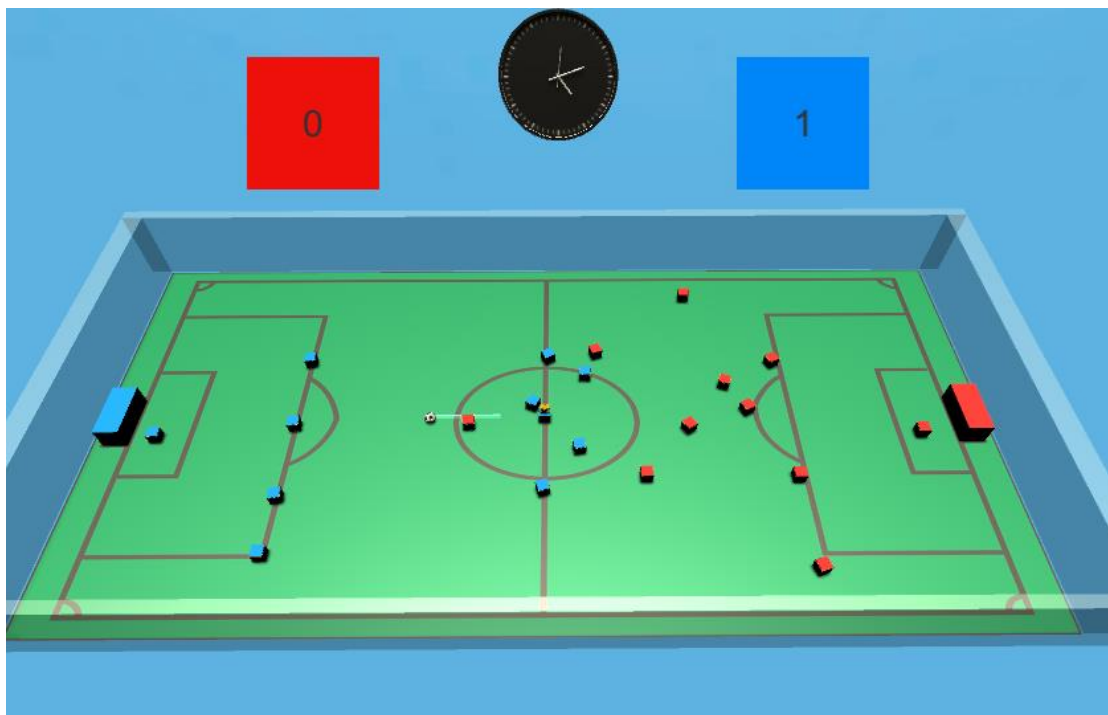
In the game, the soccer ball may fly out of the field due to the large force, which can be solved well by covering a transparent ceiling over the field.



To keep the player from losing himself in the crowd, the player controls a character who wears an X-shaped hat on his head.



A clock (from the unity asset store) and a football stuttering feature (from PocketRPG Trails) have been added.



## Conclusion

At the beginning of the project, we just wanted to make an AI that could kick a football. The AI did not have any kicking strategy. It just kicked the ball into the

opponent's goal. It was definitely a failed game, so we started looking at ways to improve it, and an excellent way to do this was to use behaviour trees instead of FSM to determine the AI's behaviour and strategy.

So we set up three different roles for the AI, corresponding to different behavioural logic, attacking, defending and goalkeeping. The game started to get interesting, but it was not enough. Since football is a team game and there is no teamwork between our AI's, we wrote relevant team strategies, such as Attacker's passing strategy and following strategy, to add some cooperation elements.

These improvements certainly add to the gameplay, but after inviting friends to play the game, I found that they could win in a short time, not because the game was designed to be too easy, but because players could play the game to gain insight into the agent's strategy and thus beat them effectively and this is where ML AI could be to step in.

The ML AI is my favourite part because no one can know what the AI's next move will be. Everything is unknown, like Schrödinger's cat. For ML agents, we started with the PPO training method, but it did not work very well. By improving it to the MA-POCA training method, the AI's started to learn teamwork and how to knock out their opponents as quickly as possible.

However, the game gets a little harder in this mode, and some of my friends lost to the AI in this mode, so I decided to add a little egg to enhance the player's game against the ML AI in a brand new mode with a giant body and sprint ability, and the feedback from my buddies was good for this improvement.

Future research directions will likely include dividing the field into strategic zones to influence AI decisions and using some of the more advanced Flocking algorithms to enable team following. In the meantime, FSM and Behavior Tree could be used in conjunction to improve the aesthetic aspects of the game. In the machine learning aspect, increasing the number of training steps can effectively optimise the performance of the AI.

## References

- A. L. Samuel. (2000). *Some studies in machine learning using the game of checkers*<https://doi.org/10.1147/rd.441.0206>
- Allerin. (2020). *The role of behavior trees in robotics and AI | artificial intelligence* /. <https://www.allerin.com/blog/the-role-of-behavior-trees-in-robotics-and-ai>
- Andrew Cohen. (2020, February 28,). *Training intelligent adversaries using self-play with ML-agents*. <https://blogs.unity3d.com/2020/02/28/training-intelligent-adversaries-using-self-play-with-ml-agents/>
- Baeldung. (2020). *Genetic algorithms vs neural networks | baeldung on computer science*. <https://www.baeldung.com/cs/genetic-algorithms-vs-neural-networks>
- Bors, M. L. (2018). *What is a finite state machine?* <https://medium.com/@mlbors/what-is-a-finite-state-machine-6d8dec727e2c>
- Colledanchise, M. (2017). *Behavior trees in robotics*.  
[https://explore.openaire.eu/search/publication?articleId=od\\_\\_\\_\\_\\_260::5d1fe75c65f431b7cfd8c98b80d4e7f4](https://explore.openaire.eu/search/publication?articleId=od_____260::5d1fe75c65f431b7cfd8c98b80d4e7f4)
- Fathy, H., Raouf, O. A., & Abdelkader, H. (2014). Flocking behaviour of group movement in real strategy games. Paper presented at the *2014 9th International Conference on Informatics and Systems*, PDC-67.
- Foerster, J., Nardelli, N., Farquhar, G., Afouras, T., Torr, P. H., Kohli, P., & Whiteson, S. (2017). (2017). Stabilising experience replay for deep multi-agent reinforcement



learning. Paper presented at the *International Conference on Machine Learning*, 1146-1155.

Hamed, A. Y. (2010). A genetic algorithm for finding the k shortest paths in a network. *Egyptian Informatics Journal*, 11(2), 75-79. <https://doi.org/https://doi.org/10.1016/j.eij.2010.10.004>

John Schulman, & Alec Radford. (2017). *Proximal policy optimization*. <https://openai.com/blog/openai-baselines-ppo/>

Lowe, R., Wu, Y., Tamar, A., Harb, J., Abbeel, P., & Mordatch, I. (2017). Multi-agent actor-critic for mixed cooperative-competitive environments. *arXiv Preprint arXiv:1706.02275*,

Panait, L., & Luke, S. (2005). Cooperative multi-agent learning: The state of the art. *Autonomous Agents and Multi-Agent Systems*, 11(3), 387-434.

*Proximal policy optimization*. (2017). <https://openai.com/blog/openai-baselines-ppo/>

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv Preprint arXiv:1707.06347*,

Schwab, B., & Schwab, B. (2004). AI game engine programming (pp. 281-290). Hingham: Charles River Media.

Sehgal, A., La, H., Louis, S., & Nguyen, H. (2019). (2019). Deep reinforcement learning using genetic algorithm for parameter optimization. Paper presented at the *2019 Third IEEE International Conference on Robotic Computing (IRC)*, 596-601.

Simonini, T. (2020). *An introduction to unity ML-*

*agents*. <https://towardsdatascience.com/an-introduction-to-unity-ml-agents-6238452fcf4c>

Tsung, D. (2016). *Don't re-invent finite state machines: How to repurpose unity's*

*animator*. <https://medium.com/the-unity-developers-handbook/dont-re-invent-finite-state-machines-how-to-repurpose-unity-s-animator-7c6c421e5785>

Vorkflowengine. (2020). *Why developers never use state*

*machines*. <https://workflowengine.io/blog/why-developers-never-use-state-machines/>

Watson, I., Azhar, D., Chuyang, Y., Pan, W., & Chen, G. (2008). Optimization in strategy

games: Using genetic algorithms to optimize city development in freeciv. *University of Auckland*,