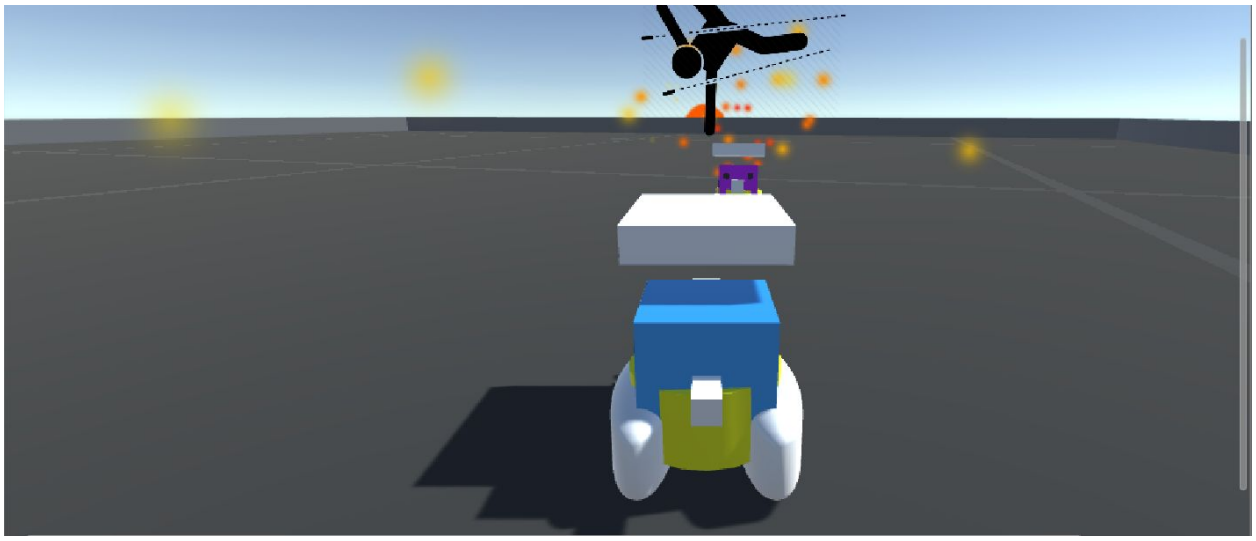
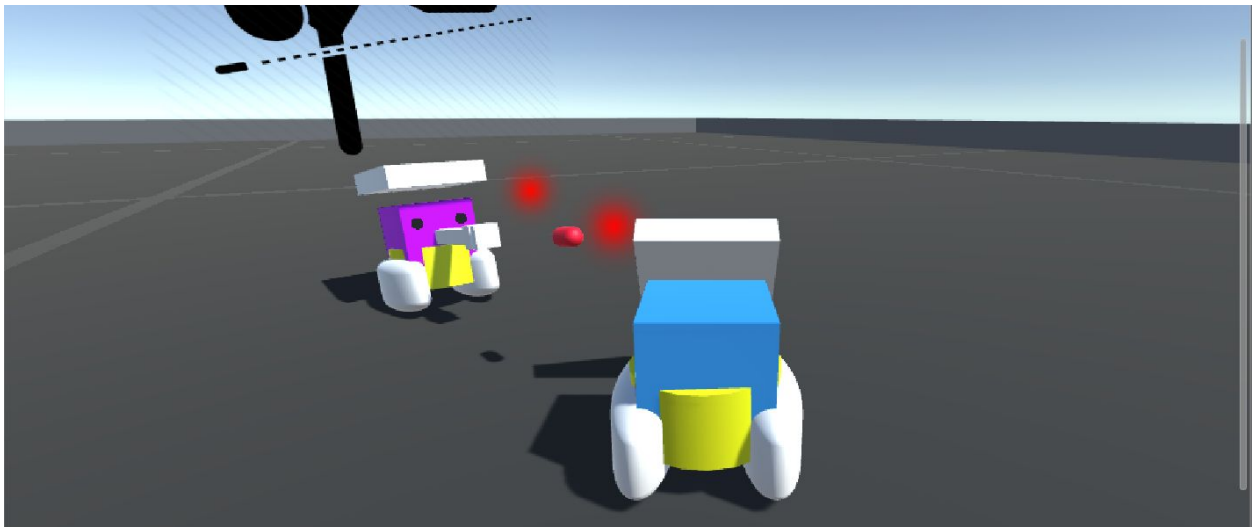
Combat Tank AI Agents

Using FSM, ML Agent trained using the FSM and Competitive Learning





Rohan Menon

1901120

MSc Computer Games Technology AI Coursework

INTRODUCTION

For the AI course I decided to write a program that generates an enemy tank AI which is able to respond to player actions and figure out how to fire and block projectiles. The player decides the movement of the tank, firing a bullet and using a shield to block an attack. The application used to showcase and train the AI is Unity with C# as the primary scripting language for Unity. For training and using the ML-Agents algorithm and to generate a usable Neural Network, I am using the ml agents package provided by Unity to use Tensorflow provided by google with Python as the primary scripting language.

The key learning for the AI should be to move towards the player and when looking at the player, it should try to shoot. The game is relatively straight forward but defining an AI that does these movements in a “non predictable” fashion is quite challenging due to the simplicity of the game.

I started by creating an FSM Agent and then trained an ML Agent using the FSM system. I also wanted to try to train the ML Agent by making it combat itself later on but that would have taken a considerably longer time.

METHODOLOGY

The following is a discussion of the actions available to the agents(both player and AIs) and the properties of each player and AI.

The actions available to each player in the game are:

- Move forward or backward
- Move left and right
- Shoot a bullet
- Deploy a shield which blocks an attack

The rules for the game are:

- Each tank has a predefined health value
- When hit by a bullet from another tank the player loses 10 health
- When hit by a bullet when shield is in front, no health is taken
- When one of the player health reaches zero, victory is achieved/training is reset

- Once a bullet is fired, a reload animation takes place during which the player cannot block or fire

The properties that are common for all players and enemy AIs are defined by an interface called “IPlayerStats.cs”.

This is the interface that also defines when a certain action occurs during gameplay, like a successful hit taking place, successfully blocking a hit, getting damaged from an enemy hit, etc.

THE AI AGENTS

The stars of the show, the AI agents are the 2 methodologies I am using to define the behavior of both AIs.

1. FSM (Finite State Machine)[Chionglo, 2019]
    a. A finite state machine is used to transition between different states and execute certain behaviours within those states.
    b. A fuzzy state machine is very similar to a finite state machine but in this case, I modify the behaviour of the agent in certain states depending on it’s remaining health but it isn’t true fuzziness
    c. My FSM is able to transition between 5 states which we shall go into more detail about in the next section
    d. The FSM will try to move close to the player, fire a shot, try to move away from the player while reloading and also block if it’s health is below a certain health threshold
2. ML Agent Trained using FSM competitor[Jun LAI, Xi-liang CHEN, 2019]
    a. An ML agent uses vector observations, actions an Agent takes and the reward result from those actions to create a mapping of actions that give it a higher reward, based on it’s observations at the time
    b. The ML agent a configuration yaml file(details of which will be given in the ML Agents section of this report)
    c. I am training the agents using Proximal Policy Optimization

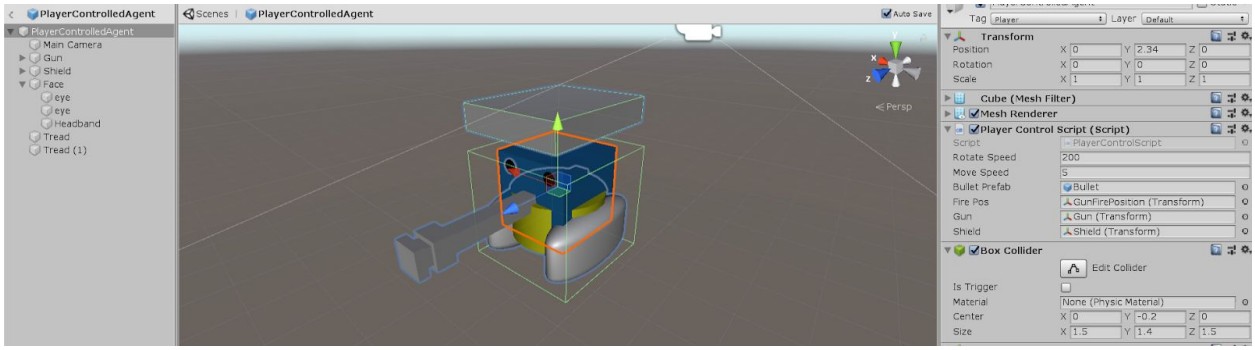
ASSETS USED

Tools used:

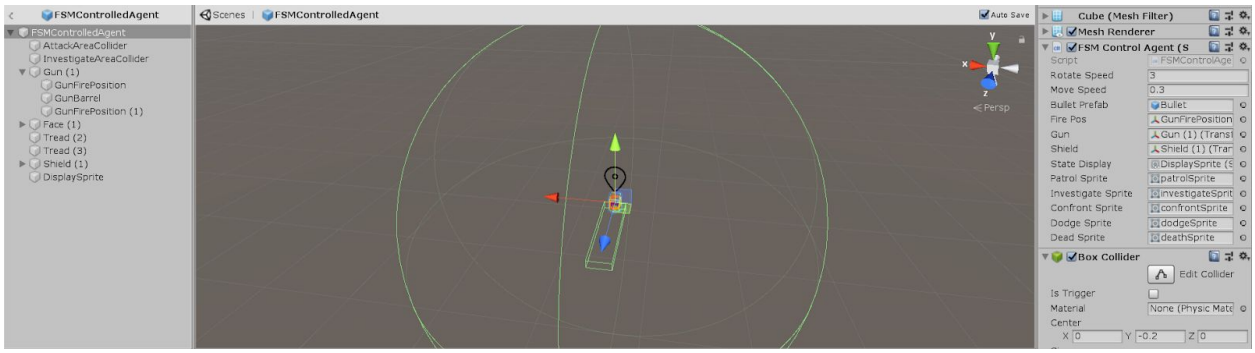
- Unity
  - DOTween (a tool used to create simple animations through code)
  - ML-Agents
- TensorFlow

Description Of Prefabs (In Trainer/Prefabs):

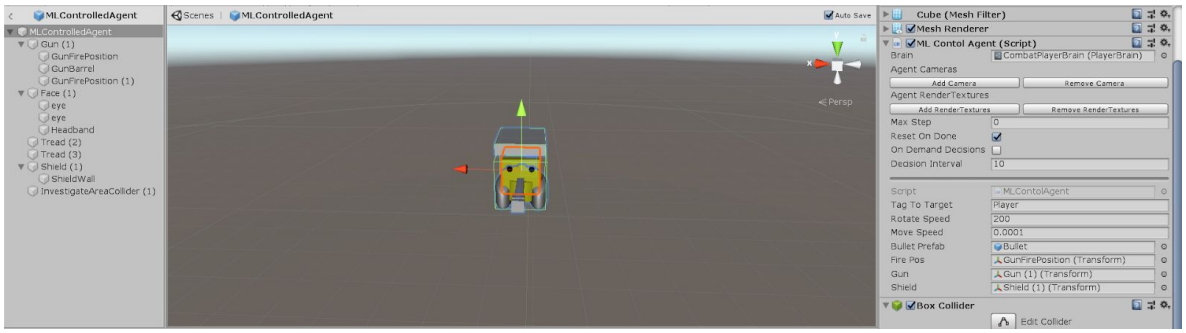
- PlayerController:
  - Gun: Gun object animating to fire a shot and the place where bullet will spawn
  - Shield: Shield object animating to block a bullet
  - Other Objects to make it look remotely like a tank
  - PlayerControlScript



- FSMPlayerController (Same as the PlayerController but with):
    - AreaCollider: The collider that is responsible to move to Confront State
    - InvestigateAreaCollider: The collider that is responsible to move to Investigate State
    - StateSprite: Sprite that shows what state the FSM is in right now
    - FSMControlAgent script instead of PlayerControlScript



- MLPlayerController(Same as the PlayerController but with):
    - InvestigateAreaCollider: The collider that is responsible to keep track of enemy player
    - StateSprite: Sprite that shows what state the FSM is in right now
    - MLControlAgent script instead of PlayerControlScript



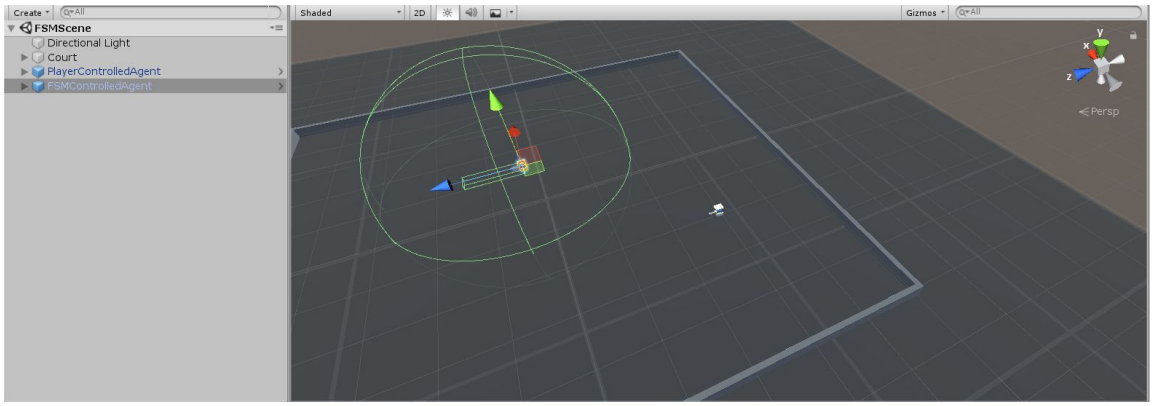
- Bullet:
    - Just a projectile that will move forward as it has spawned with a few particle effects
    - On collision with a wall or other agent, it will explode and emit a few particles
    - When colliding with a player or shield, it will call the ShieldHit, OnShieldHit functions respectively

Scripts that are important(In Trainer/Scripts):

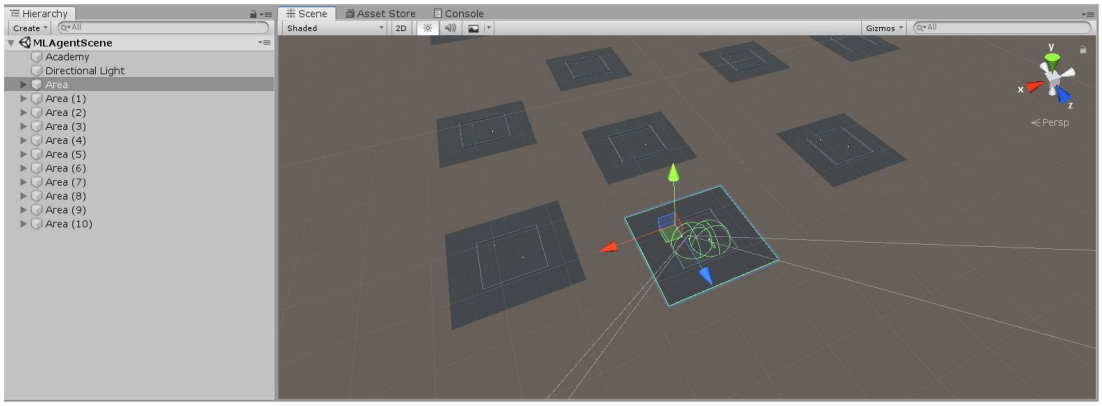
- PlayerControlScript
- FSMControlAgent
- MLControlAgent
- BulletScript
- InvestigateAreaScript
- AttackAreaScript
- InvestigateMLScript

Scenes Used(In Trainer/Scenes):

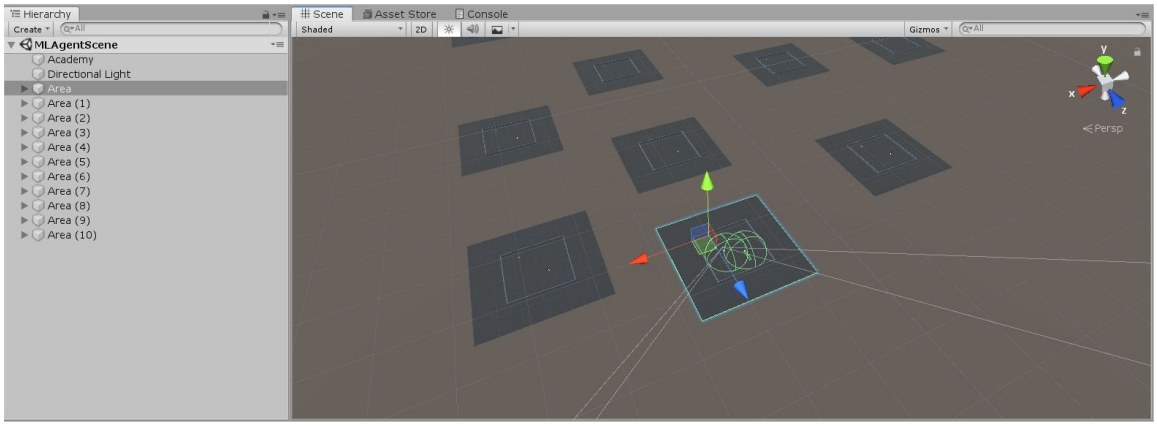
- FSMScene: Scene used to play against the FSM
    - Only has a FSM agent and a player agent
    - Has walls surrounding it



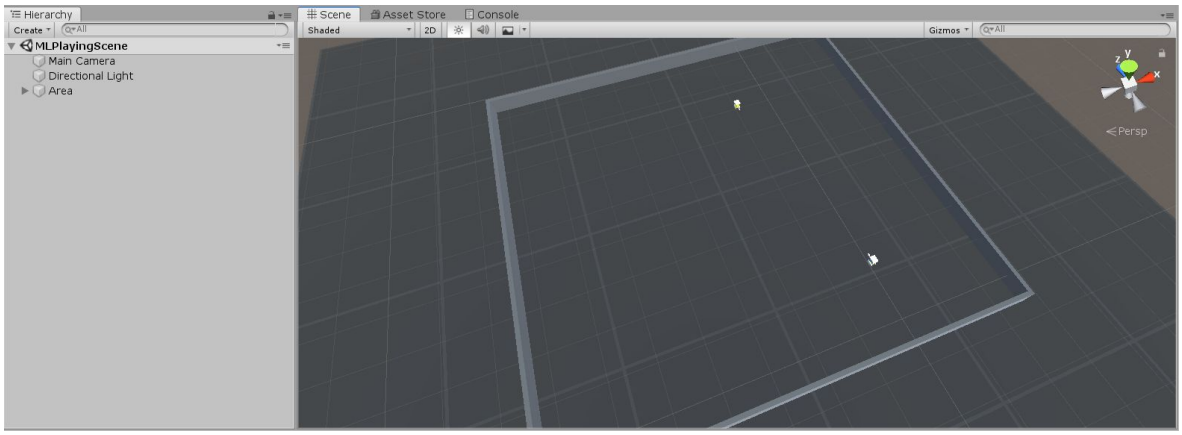
- MLTrainingFSMScene: Scene used for training against FSM
    - Has 10 copies of an ML Agent playing against an FSM state
    - This is what is used to train the AI against the FSM



- MLTrainingMLScene: Scene used for training against FSM
    - Has 10 copies of an ML Agent playing against an FSM state
    - This is what is used to train the AI against the FSM



- MLPlaying: Scene to play against trained ML Agents
    - This is what is used to play against the ML Agent



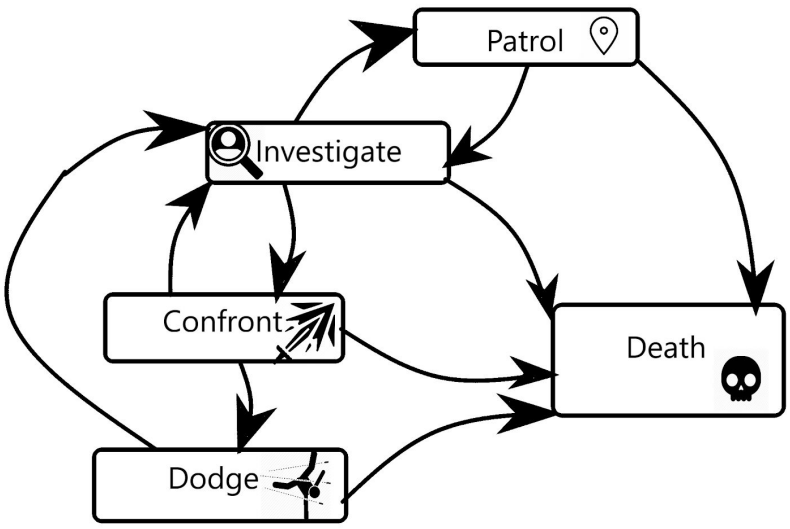
FSM (FINITE STATE MACHINE)

I have written 5 functions related to my state machine.

- OnSwitchState() gets executed when transitioning into a new state
- OnExitState() gets executed when transitioning from 1 state to another
- OnProcessState() gets executed every frame for the current state
- OnEntryState() gets executed when switching to a new state for the first time and changes the stateDisplaySprite as well
- CheckState() calls OnProcessState() in every fixed update and OnExitState(State oldState) and OnEnterState(State newState) if transitioning to a new state

My FSM system has 5 states

- Patrol
- Investigate
- Confront
- Dodge
- Dead



Finite State Machine
State Transitions

Patrol state:

On entry into the patrol state, the agent defines a set of 3 waypoints near it and then tries to loop/move within those 3 waypoints.

When an enemy agent enters the investigation collider, it switches state to Investigate.

Investigate state:

On entry into the investigate state, the agent stores that a target has been acquired.

If it isn't blocking and it's health is below a threshold, it will try to block as well.

When the target is acquired, it will always try to move towards the target.

During this time it checks the AttackArea collider and moves to Confront state if the target is in front of it.

Confront State:

On entry into the confront state, if the agent is blocking, it will unblock so that it can fire.

Now instead of targeting on the target, it will target a point based on the targets velocity as well to achieve higher accuracy on moving targets.

If the target is within a certain range of accuracy, the FSM agent will fire a shot, and then transition to the Dodge state.

Dodge State:

On entry into the Dodge state, the agent will stop so that it can start reversing

While in dodge, it will try to put it's shield down if it isn't reloading and also try to move away from the target to avoid retaliation

Once reloaded it will switch back to investigation or patrol based on whether or not there is a target to investigate

The fuzziness in this State Machine is achieved by the FSM checking it's remaining health

and orientation to decide if it should deploy it's shield or not. Which allows for some amount of variety in gameplay.

ML Agent trained using FSM competitor (First Attempt)

The basics of ML Agent Training involves 3 objects:

- Academy: Responsible for training and inferencing brains
- Agent: Responsible for storing observations, doing actions based on brain output and calculating a reward for the actions it takes
- Brain: The Neural Network generated for training and to feed in input controls to the agent

Let's discuss what I am using for the agent:

Actions:

In my MLAgent, AgentAction(which responds to an agents decision) will set a bool value based on each vector input(UpDown, LeftRight, Shoot, Block).

This bool value for each vector input will decide what action is taking place in the FixedUpdate loop(For example onDecisionBack, onDecisionForward, etc).

My vector input actions are:

- UpDown: Gives forward motion when float is greater than 0.1, gives backward motion if less than -0.1 and no movement if between -0.1 and 0.1
- LeftRight: Gives left rotation when float is greater than 0.1, gives right rotation if less than -0.1 and no rotation if between -0.1 and 0.1
- Block: Simulates block button being pressed if greater than 0.5, block button not being pressed if less than 0.5
- Shoot: Simulates shoot button being pressed if greater than 0.5, shoot button not being pressed if less than 0.5

Observations Collected:

Observations are used by the ML agent to store a snapshot for the environment to draw inferences as to which actions to take actions it has taken.

I considered using the ray perception algorithm provided by unity but I think for my sample space, I didn't really need it and it would muddy the observation maps too much.

In my case I am storing the following values:

- Agent Health
- Agent isReloading
- Agent isBlocking
- Agent transform localPosition
- Agent transform y rotation(because it is a 2D game)
- Agent rigidbody velocity.x
- Agent rigidbody velocity.y
- Whether or not there is a target
- Target health(0 if no target)
- Target isReloading(false if no target)
- Target isBlocking(false if no target)
- Distance to Target(0 if no target)
- Target localPosition(0 if no target)
- Amount to rotate to look at target - lookAtDeviation(0 if not target)
- Target transform y rotation(0f if no target)
- Target rigidbody velocity.x
- Target rigidbody velocity.z

Rewards Collected:

Positive Rewards:

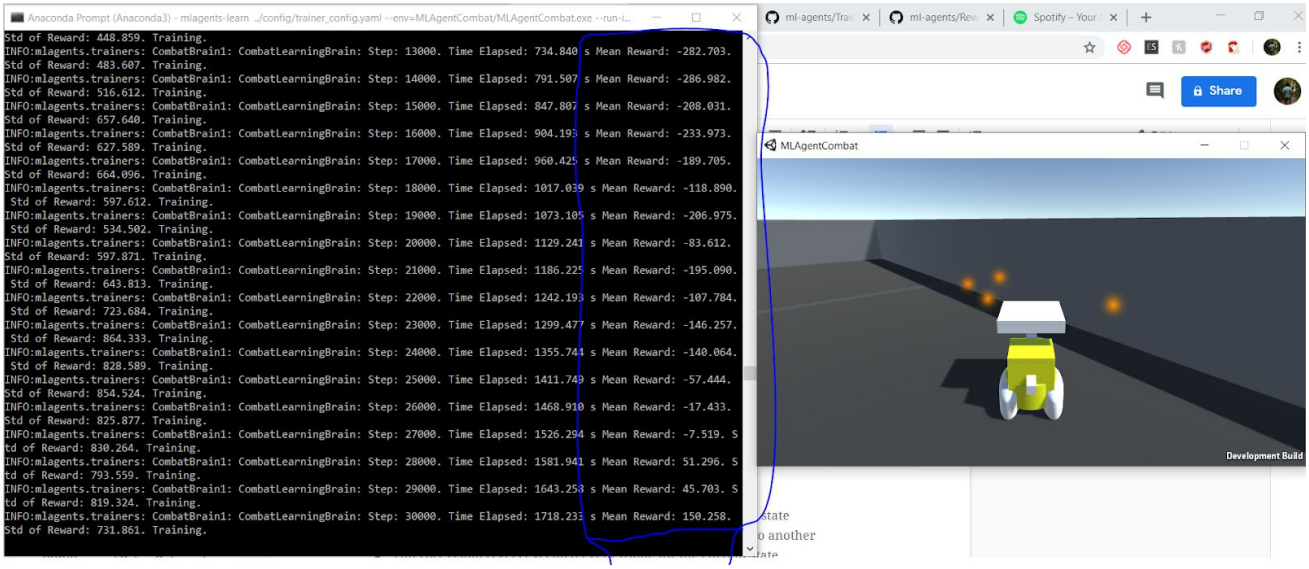
- On Moving Forward + 0.001
- On Moving Back + 0.000000001
- Looking within 30 degrees of player = 0.01 * how far away it is looking
- Looking within 10 degrees + 2
- Firing when looking at enemy + 5
- If closer to the enemy than distance threshold + 0.1 * distance
- On Attacking + 5
- On Successful hit + 100
- On Successful Block + 2
- On Shielded Hit + 25
- On Killing Enemy + 200

Negative Rewards:

- If not moving forward or backward - 0.00000001

- If firing when not looking within 10 degrees of player - 0.1
- If trying to fire without target - 0.5
- If not looking at enemy - 0.00001 * deviation from aim
- If far away from enemy - 0.001 * distance to target
- If trying to fire when reloading or blocking - 0.00001
- On Getting hit - 25
- If Blocking when reloading - 0.0001
- OnDeath - 400
- When losing track of player - 1
- If colliding with wall per frame - 0.1

PPO(Proximal Policy Optimization) to train my ML Agent:



The picture above shows the ML agent training and the general increase in reward over time.

I am using a PPO(Proximal Policy Optimization to train my ML Agent).

“PPO uses a neural network to approximate the ideal function that maps an agent's observations to the best action an agent can take in a given state. The ML-Agents PPO algorithm is implemented in TensorFlow and runs in a separate Python process (communicating with the running Unity application over a socket).” [Unity Technologies, 2019]

I had enabled curiosity learning because I want the ML agent to be able to check if doing something differently allows it a greater reward rather than being trained in a set way as soon as it gets the first set of good rewards.

My YAML config has the following parameters:

```
trainer: ppo
batch_size: 1024
beta: 5.0e-3
buffer_size: 1024
epsilon: 0.2
hidden_units: 128
lambd: 0.95
learning_rate: 3.0e-2
learning_rate_schedule: linear
max_steps: 750000
use_curiosity: true
memory_size: 256
normalize: false
time_horizon: 100
sequence_length: 100
summary_freq: 1000
use_recurrent: true
reward_signals:
    extrinsic:
        strength: 1.0
        gamma: 0.99
    curiosity:
        strength: 0.02
        gamma: 0.99
        encoding_size: 256
```

I shall go through some of the important values above: [Unity Technologies, 2019]

- Batch_size: The number of experiences in each iteration of gradient descent.
- Buffer_size: The number of experiences to collect before updating the policy model.
- Beta: The strength of entropy regularization
- Epsilon: Influences how rapidly the policy can evolve during training
- Lambd: The regularization parameter
- Learning Rate: The initial learning rate for gradient descent
- Max_steps: The maximum number of simulation steps to run during a training session
- Use_Curiosity: Allow it to make random curious decisions based on actions it hasn't taken
- Time_horizon: How many steps of experience to collect per-agent before adding it to the experience buffer
- Num_layers: The number of hidden layers in the neural network
- Reward Signals: The reward signals used to train the policy. I enabled Curiosity here.

The curiosity Reward Signal enables the Intrinsic Curiosity Module. This is an implementation of the approach described in "Curiosity-driven Exploration by Self-supervised Prediction" by Pathak, et al. It trains two networks:

- an inverse model, which takes the current and next observation of the agent, encodes them, and uses the encoding to predict the action that was taken between the observations
- a forward model, which takes the encoded current observation and action, and predicts the next encoded observation.
- The loss of the forward model (the difference between the predicted and actual encoded observations) is used as the intrinsic reward, so the more surprised the model is, the larger the reward will be. [Unity Technologies, 2019]

Learnings to take forward to the next phase:

Ray perception was muddying my results quite a bit. I decided to use a simpler AI that is only keeping track of a target objects position.

I realized that I was passing continuous input to the ML Agent rather than discrete values which significantly increased the complexity of the AI Vector Actions. I switched to using **Discrete** vector inputs since my Agent was already built for explicit button presses anyway.

I then noticed that the **observations of the ML were very similar**, this is due to the **inherent predictability of my FSM agent**. I then decided to move onto phase 2 which is training my ML agent vs another ML agent.

ML Agent trained using ML Competitor (Second Attempt)

I wanted this training session to go through as many iterations as possible so that it can properly train it's environment variables to the actions it needs to take. Due to this I started training the ML Agent against another MLAgent. This will allow for far greater variety in input vectors and allow the ML to train a lot better. [João, 2019]

My actions were the same as the original ML but I was now using Discrete input actions rather than Continuous actions.

My Observations were also the same as the last experiment

Also I realised that using curiosity can muddy the results if there are multiple increments of very small rewards so I reduced the number of reward modulation as well. So I reduced the frequency of rewards too.[Arthur, 2018]

My new reward set was:

- Positive Rewards
    - If firing when looking at within 2 degree of target +5
    - If firing when looking at within 20 degrees of target +2
    - If looking at target within 10 degrees of target +0.004
    - If looking at target within accuracy threshold +(0.001 * distance/accuracyThreshold)
    - If distance within threshold Add Reward +0.1

    - On successful Firing +3
    - On killing player +800
    - On successful hit +300
    - On successful block +2
    - On shield hit +25
    - If track a target +0.001
    - On killing an enemy +500
- Negative Rewards
    - If firing when target greater than 20 degrees - 0.1
    - If firing without target in sight -0.5
    - If looking away from player - 0.01
    - If no target -0.001
    - If firing incorrectly -0.001
    - On being hit -100
    - On Incorrect block -0.001
    - On Death -800
    - If target exists collider -1
    - If colliding with wall -0.1f

Due to training it against an ML, I also wanted to add more layers to the neural network, so that it can form greater observations from the results.

This was my new YAML file

```
trainer: ppo

batch_size: 256

beta: 5.0e-3

buffer_size: 2048

epsilon: 0.4

hidden_units: 128

lambd: 0.95

learning_rate: 3.0e-2

learning_rate_schedule: linear

max_steps: 500000

use_curiosity: true

memory_size: 256
```

```
    normalize: false
    num_epoch: 3
    num_layers: 6
    time_horizon: 10000
    sequence_length: 200
    summary_freq: 10000
    use_recurrent: true
    reward_signals:
        extrinsic:
            strength: 0.9
            gamma: 0.99
        curiosity:
            strength: 0.001
            gamma: 0.99
            encoding_size: 256
```

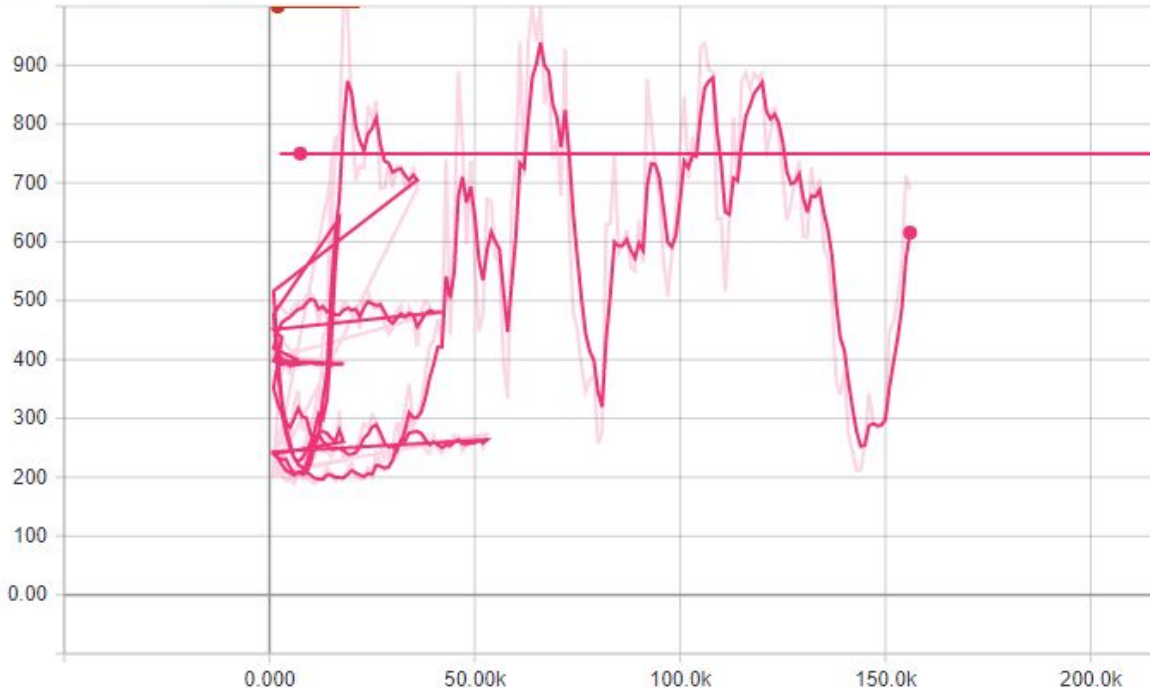
The ML Agent vs ML Agent simulations performed a lot better than the FSM vs ML Agent simulation do to the much wider sample set to train on.

DATA

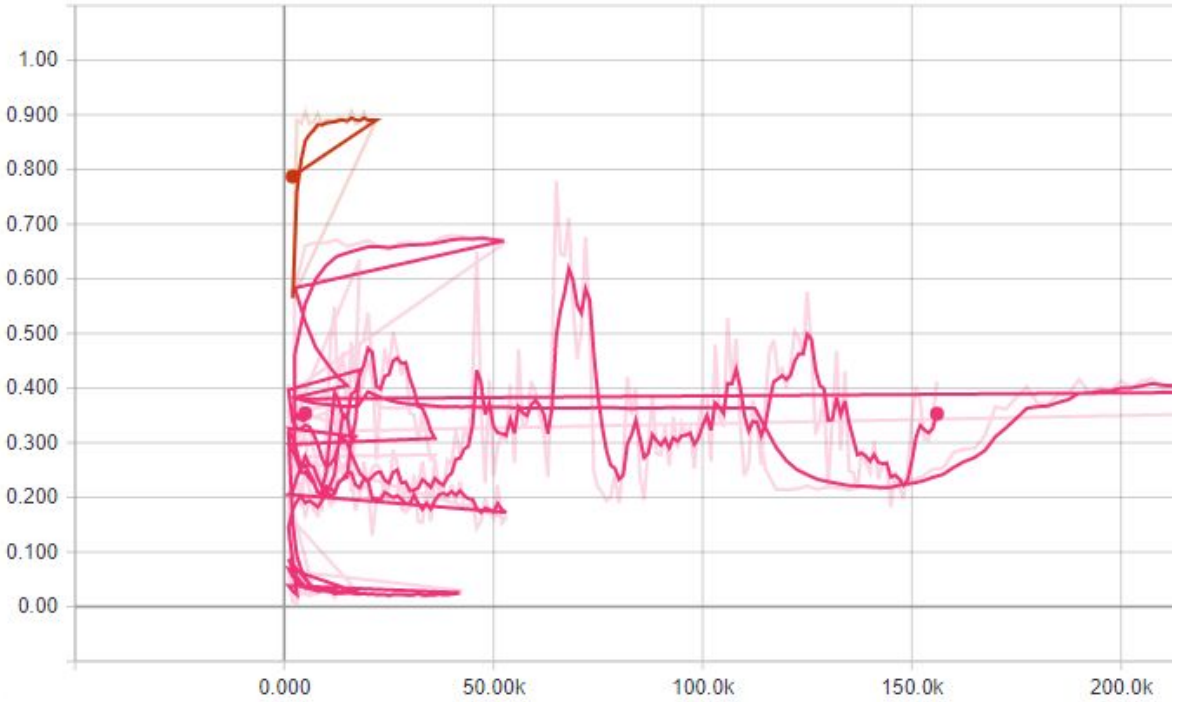
My ML agent was getting too many negative rewards and was not able to properly learn the rules of the game. As you can see it isn't able to concisely measure which actions are a good result due to it getting interference from the negative rewards that have been setup. It will still be able to learn if I want to keep it going for more steps, but it makes more sense to refactor the reward algorithm and make the ML Agents learning environment less muddy.

Below, the orange is the training that was done with the FSM and the pink is the training done with the ML Agent. As you can see there are a lot more potential changes in the environment when fighting against an ML agent because the environment observations hardly change with an FSM.

Environment/Episode Length



Losses/Policy Loss



RESULTS

FSM:

When playing against the FSM controlled bot, although it shows a nice simulation of attacking, defending and patrolling around, it is very predictable in it's behaviour.

A player will be able to understand exactly what it does within the first 4-5 plays and be able to counter it's actions.

Challenge in a game for this kind of enemy can be added by adding more states and taking decisions between those states as well.

Although this AI took me a lot less time to implement, it is not as satisfactory as a true complicated AI. And will always have some amount of "predictability" that is easy to figure out even if I make it more complex.

ML Agents:

This predictability is what causes an issue when training the ML Agent against an FSM, the **environment inputs** are too straight forward and will lead to predictable actions for the same actions.

When playing against an ML Agent with sufficient training, the game does become significantly harder. But my training process needs to be changed considerably. I had a few successes but the way I was assigning rewards in most cases did not lead to the agent learning properly.

But a training algorithm with accurate rewards is hard to define. In my case I had to go

through multiple steps of very long training to achieve an averagely successful AI.

But the algorithm was definitely learning and understanding what to do. This gives quite a lot of scope for improvement in the algorithm, especially if I tweak the rewards it is getting at the same time as well.

But in most short training results, the ML Agent was not as skilled as the FSM but developed its own personalities like:

- Blocker: Usually very good at blocking shots, but does not fire often(by adding a higher reward to a successful block)
- Attacker: Usually very good at shooting but not good at blocking(by adding a higher reward to successfully shooting)

This can be extremely useful for being able to generate AIs that are good at different things and creating variety in gameplay. I would like to explore this avenue further in future research.

CONCLUSION

I learnt that if you **use curiosity**, you should not try to add very many small rewards as the curiosity function will never pick up **small increments of reward**.

If simulating key inputs, a **Discrete** input is a lot easier to train than **Continuous** input.

In conclusion, an FSM state machine in a game like mine is a lot easier to implement than an ML Agent, but lacks any true Artificial Intelligence and becomes very predictable.

Training an MLAgent against a highly predictable FSM will not provide a good neural network for the agent as it only knows how to respond to the predictable actions of an FSM AI. Hence, training between 2 ML agents will provide a lot better input for the learning environment due to randomized movements.

But given enough time for training, and just by **tweaking the reward parameters**, an ML agent will **not only be better at providing realistic gameplay, but can also be used to create behaviours that will be different from each other**, relatively quickly (But not as fast as directly coding an FSM).

FUTURE RESEARCH

I want to be able to create a set of rewards to be tweaked so that I an output agents with varying behaviours and try to compare my results over long sets of training. I am pretty sure the end result will be interesting to see as it will lead to multiple personalities of enemies with the same tools but with a high variety in their behaviour.

REFERENCES

Cornell University, Arthur Juliani, Vincent-Pierre Berges, Esh Vckay, Yuan Gao, Hunter Henry, Marwan Mattar, Danny Lange (Sept 2018)

Unity: A General Platform for Intelligent Agents, Available At:

https://arxiv.org/abs/1809.02627

Unity Technologies (2019),(#2786)

Training with Proximal Policy Optimization Readme, Available At:

https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-ML-Agents.md

Unity Technologies (2019),(#2729)

Training with Proximal Policy Optimization Readme, Available At:

https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-PPO.md

Unity Technologies (2019),(#2729)

Reward Signals Readme, Available At:

https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Reward-Signals.md

J. Chionglo (2019),

Dynamic and Interactive State Machines, Available At:

https://www.academia.edu/37928827/Dynamic_and_Interactive_State_Machines

Immersive Limit (Jul 30, 2019),

Using ML Agents Tutorial Video

https://www.youtube.com/watch?v=axF_nHHchFQ

Jun Lai, Chen Xiliang, Xue-zhen ZHANG(May 2019),

Training an Agent for Third-person Shooter Game Using Unity ML-Agents

http://www.dpi-proceedings.com/index.php/dtcse/article/viewFile/29442/28424

João Ramos (Mar 2019)

Deep Reinforcement Learning using Unity ml-agents

https://towardsdatascience.com/deep-reinforcement-learning-using-unity-ml-agents-8af8d407dd5a

Arthur Juliani (June 26, 2018)

Solving sparse-reward tasks with Curiosity

https://blogs.unity3d.com/2018/06/26/solving-sparse-reward-tasks-with-curiosity/