Artificial Intelligence – Final Report

Assessing the ability of a behaviour mimicking agent to recover after unobserved factors alter its state.

Team members

Jakub Szulwinski: S1627131

Interested in: Machine learning in games
Skills: Unity3D, C#, C++, JavaScript
Wants to learn: Training NPCs in games

Fotios Spinos: S1625069

Interested in: Machine learning,
Skills: Unity3D, C#, C++, JavaScript, Math skills
Wants to learn: Machine learning, fitness functions, agent observations (the input fed to reinforcement learning agents).

William Donald: S1632091

Interested in AI: basic computer opponents in simple games
Skills: Unity3D, C#
Wants to learn: The basics of Agent creation

Caleb Ross: S1626968

Interested in: Machine learning / training agents.
Skills: Unity, C# and C++
Wants to learn: Integrating machine learning into a project and agent training techniques.

Abstract

AI is used more and more as a tool to find solutions to problems and automate certain actions that were not possible before. While they all vary in importance and impact on the field, they all provide significant improvements to new and existing applications. Additional research can benefit the entire field of AI as a whole. One of the things that can benefit greatly from machine learning is video games. Traditional AI approaches like finite state machines and decision trees are capable of producing reliable and believable agents. However modern games and real world applications require that the problem to be solved is generalised. The strongest aspect of machine learning is that agent actions are defined from used data instead of explicitly programmed instructions. Machine learning seems like a more suitable solution, because the agent actions are defined from user data instead of explicitly programmed instructions. This is why we decided to research deep learning as a solution to improving AI agents movements. Our research question therefore is:

Can an AI agent mimicking the movement path of another entity use deep Q-learning to recover from unexpected interventions diverting it from its learned path?

We will try to answer the question by creating a simple 2-dimensional (2D) environment and use deep Q-learning to “teach” a physics-based “car” to mimic another vehicle with a predefined behaviour. These states would be position, rotation and velocity, which have been previously supplied to it from user input. Then we will deploy a series of tests that will assess the performance of the agent with each iteration, and based on those results, we will improve the agent further. By the end of the investigation we should be comfortable enough with deep Q-learning to answer the question. We will then be able to judge how much more can be learned from this technology and how it could be used in a game or other aspects.

By the end of the project the agents could mimic the movement path of the dummy car. While following the dummy car, agents were keeping up with it the majority of the time. After the agent was set off course, it demonstrated its ability to recover. The agent has shown to be able to recover from mild manipulation, however with big manipulations, it proved to be a problem for the agent's ability to recover.

Contents

Proposal

Research Question

Can an AI agent mimicking the movement path of another entity use deep Q-learning to recover from unexpected interventions diverting it from its learned path?

- Recovery suggests that the agent will be able to maximize its fitness after an unknown factor originated from the environment forces the agent to behave undesirably.

- A state refers to the position, rotation and velocity of an entity.

- Unexpected interventions refer to events manipulating the entity's state without being used as input to the agent. In other words, the agent is unaware of these events.

What are we going to learn?

By the end of the investigation a deeper look into deep Q-learning being used to simulate entity movement behaviours should hopefully provide a measure on how useful such a technology could be in a commercial game, and how such a technology could further be used in other aspects of game development such as procedural animation. As mentioned in the Ubisoft Montreal paper, using the combination of an animation database and machine learning, animation frames can be interjected as a “best fit” situation to allow for more dynamic motion.

However, this has the problem of requiring a very large animation database and is therefore optimized by precomputing any potential motion that is matched between the animation database and the animating entity. When matched with the idea’s covered in this project, potentially such a system could be created with a much smaller animation database footprint by further using deep Q-learning to drive the kinematics chains of an animation and not rely on learning hours of motion capture data.

Project outline

The main implementation of the project will be made using the Unity game engine, as it provides a stable environment with built in physics and a plethora of tools to support development. Two types of scenes will be created: One for training the agent and another for launching the agent into different environments and assessing limitations or advantages it offers.

To reduce the complexity and show the general applicability of the project, deep Q-learning will be deployed in a 2-dimensional (2D) environment and applied to a physics-based “car” entity focusing on the simple movement states position, rotation and velocity, which have been previously supplied to it from user input.

Half-way Stage

By the half-way stage of the project a solid ground work should hopefully be laid out to provide better iteration at later stages. As mentioned before, the main implementation will use a physics-based “car” entity which will need to be simple and provide an easy way to access it’s steering and driving capabilities from any external script in the project to allow the deep Q-learning algorithm to take control of it. Since the aim is to take premade data and learn from it a movement state recorder will need to be created in order to record imputed data from a user, the movement data recorded from the user includes the position, rotation and velocity several times per second to provide reliable and usable data. This data will then need a system to serialize it and store it externally for later use as a XML file.

For training the agent, a scene void of any interaction is required to learn from a data set using deep Q-learning by accessing an external python package (ML-Agents). This scene should also include a visual indicator of the path that the agent is learning. This should be shown as another coloured car and/or path.

For testing the agent, another scene will be required to be made in order to provide a testing ground to test random interactions triggering forces to the agent to test the agent’s ability to return to the desired movement. This scene will require

A wide range of experimentation will also need to have been taken by this point to ensure a “best fit” solution can be found for the deep Q-learning algorithm. These tests may include different reward calculations and/or punishments. Following these experiments some testing should take place to assess the training of the agent such as the difference between the initial user data and the trained agent and the agent’s overall fitness throughout runs.

Training Scene

The training scene provides an entity with a fixed movement path and the reinforcement learning agent attempting to mimic it. The representation of the two objects will be a car as it’s shape can be simplified to a single object such as a rectangle. This approach simplifies the development of the product while producing a clear conclusion to the research question noted earlier in the document. Similarly the approach presented in this paper can be used by AI agents controlling multi-hierarchy shapes. The human body for instance consists of a variety of individual components with the ability to move independently. The inclusion of complex shapes exceeds the scope of the paper and produces a landscape for future research.

The entity with the fixed movement will originate from recording the state of a physics based object. This approach eliminates fitness errors occurring from animations unbound by physics. Animations presenting an unrealistic motion cannot be fully copied by a physics object. This suggests that maximum fitness may never be achieved.

Training Process

The 2 training phases will utilize faster simulation updates (Higher time scale) in order to perform more training iterations in a shorter amount of real-world time. The first training phase will encourage the agent to follow a fixed set of rules.

To assess the agent’s ability to mimic an entity’s movement, the software will rely on the Euclidean distance between their global position, rotation and velocity. The fitness function will offer a larger reward if the difference between the desired and current object state is low. Accordingly a lower reward (punishment) suggests that the agent does not mimic the car entity accurately.

Once the agent’s average fitness surpasses a set value, the second training phase will begin. This phase will be used to teach the agent how to recover from unexpected events. These unexpected events will be randomly triggered in order to allow the agent to recognize that unpredictable events are possible and that it will have to recover to an ideal movement state.

Testing Scene

The testing scene will launch the trained agent to an environment designed for data gathering and evaluation. Unlike the training scene, the testing scene will be updated and rendered in real time to demonstrate that the agent has learned the entity’s movement behaviour to the user.

A visual depiction of the trained agent will follow the path it has learned alongside the original “Car” that it was trained to mimic. The user can then use inputs such as key presses in order to trigger the events that will manipulate the agent (e.g. Applying a force to one side of the agent).

Technology

Deep Reinforcement Learning Technique

Deep Q-learning: A deep reinforcement learning technique that allows us to observe an environment and define a fitness function to make the agent evolve through data passed to the agent. A key component of the technique is trial and error, as the agent recognises which actions result in a positive or negative reinforcement through rewards and punishments administered on each iteration. deep Q-Learning is optimized to pick the actions leading to the higher rewards. As the agent explores the environment further it improves at identifying which actions lead to maximum rewards.

Development Tools

Unity is a readily accessible game engine with built in tools such as physics simulations that will cover the needs of the project. This will allow us to focus on the creation of the agent and environment. An essential advantage is that the engine supports C# which is a language familiar to all authors working on the project and provides a foundation to iteratively work on the project more effectively. The game engine’s robust interface combined with the programming language of choice provides a convenient and highly customizable way to modify an object’s properties and it’s attributes to make agent modifications more user friendly.

Another benefit of using unity as our base, is that it provides an addon package known as “ml-agents”, which provides us with a variety of classes allowing the communication between C# and Python code containing the deep Q-learning algorithm. The tool includes a variety of virtual methods used for passing inputs and retrieving outputs originated from the agent. To further aid development, the package also provides a separate application known as “Jupyter notebook” which is used for displaying the agent’s fitness recorded from the latest training results so that data can be gathered from each generation of the agent.

Evaluation

To come up with a reliable conclusion, a comparison will be made between the fitness levels recorded from the first and second phases of training. The comparison will allow us to calculate the difference in fitness levels at specific points in time, along with assessing the agent’s ability to return to the state producing the best recorded fitness level.

To produce more accurate results we will also calculate the average fitness recorded from multiple runs over a specific time period during the second phase of training (The outcome will then be an average fitness over time graph). The average fitness will provide an overall evaluation on the agent’s ability to return to its usual state. This is critical as external factors trigger randomly in the environment. To gain additional insight on the agent’s movement behaviour the paper will provide graphs comparing movement state elements (position, rotation, velocity) recorded in the first and second phase.

Another consideration is to estimate how fitness levels vary in multiple runs. This measurement will provide a clear view on the average amount of iterations / time required to train the agent. Through this, an insight on applying some of the fundamentals in this project to another should be clear and easy to identify.

Software Architecture Design

To get the agent to mimic the car behaviour and be able to recover to its original path, certain mechanics had to be designed. This is used to train the agent, record its movement, reward it and run environment events.

Car movement

The project is going to make use of two cars. One of the cars is going to demonstrate a fixed behaviour. This vehicle (dummy car) will read the contents of the file and apply the described states to itself. The contents of the file will represent the animations the dummy car will strictly follow. The other car will be a physics object and will allow the retrieval of inputs through methods. This car will retrieve these inputs and turn them into actions. This is very significant as inputs can be retrieved from the user for testing and recording animations or from the machine learning algorithm during training. These two cars and a platform make up a training environment. The machine learning algorithm controlling the car will attempt to mimic the data retrieved from the agent car in the current frame.

The controllable vehicle will provide methods for setting the amount of torque (to rotate the object left and right) and the amount of forward force applied to the object. This is going to be determined through getting the horizontal and vertical axis. Script is also going to get velocity, rotation, position and max speed from the object.

DummyCarController script will move the vehicle along the predefined path, following fixed instructions. It will get the path from an XML file and move the vehicle along it. The plan is for the UpdateController method to take data from CurrentPathList, which takes it directly from the XML. Then it should set the vehicle position and rotation to match the data from the path. ResetVehicleController method will choose a random animation and a random point in the selected animation and updates a controller once a new index is being set.

Training

Training is a stage in which an agent is going to learn how to correctly mimic the dummy car movement. It is going to feature training mechanisms that judge the performance of the agent through the reward system and improve it in next iterations. TrainingEnvironmentMaster is a script that will handle methods for the environment in which training occurs. After the agent is initialized its position is going to be set to the position of the dummy car. The second script, AgentCar.cs is going to be used for the training process. It will start with referencing the agent and the dummy car and then setting maximum position distances and class attributes. In the AgentAction method reward value is going to be reset on every update. Velocities have to be increased so the agent's speed can match the dummy car. Steering and thrust are going to be applied to the vehicle so it can position itself correctly. Distances in positions between dummy car and agent are going to be calculated as well as rotations. The first approach we tried, which included rewarding the agent for getting closer to the dummy car in both position and rotation. If the distance between the vehicle was too great, then the reward value was subtracted. In the next attempt, the approach was taken to reward the agent as long as the agent would not exceed a specified difference in states. A reward will then be given on each frame. Those rewards are later going to be used to assess performance of the agent. This feature is important because it is going to allow us to judge fitness levels and introduce better updates. In the next method, agent and dummy car information such as position, rotation, velocity and distance are going to be passed. In the method AgentReset the animation state is reset and the agent position, rotation and velocity will be applied back to dummy car values.

Recording Animations

The fixed behaviour executed by the dummy car will be represented by a collection of fixed animations. Displaying animations using game objects will involve the manipulation of the object's position and rotation. As It would be unreasonable to create animations by hard coding object attributes, the software will provide functionality for recording and storing a collection of positions and rotations from a game object to an xml file. The recorder script collecting the data during runtime should allow the developer to start and stop recording through providing access to class attributes or alternatively through function calls. That would mean that calling a start or end named function would start or stop the collection of data being held by the recorder object. The recorder should not provide any functionality for storing data into a file. Instead the object should use a FileReaderWriter script to store the contents into a file when recording stops. The script should provide functions for storing and reading a collection of data into or from a file. The data stored should be passed to the FileReaderWriter to execute the desired behaviour.

Movement of the cars needs to be recorded and saved, the data can later be used to assess performance. Recorder.cs script will be used for that. Position, rotation and velocity of the agent are going to be saved in an XML file. RecordState will be used to launch the recording and StoreRecordedData stops the recording and stores gathered data to the XML file. It will be later called in the RecordingEnvironmentMaster script.

The recording feature is essential in terms of assessing the performance of the agent and progress in training.

XML ReaderWriter

The data about paths and animations collected through the process of recording will be stored in XML files. These files need to be accessible and easily manageable. All the XML files are going to be managed by the XMLReadWriter script. The methods written there will be used to save and load data in other scripts. Save method will use XmlSerializer and FileStream to create XML files with the data. Load method reads the data from these files. LoadAll method can be used to load lists of agent states. The script will also be able to switch between several paths for the car. Handling files in this way is going to ease the process of saving and loading data through calling the methods from the manager in other classes.

Environment Events

ChasingBox script will provide the functionality followed by chasing boxes and provides the means of changing the agent's state. The plan is for the script to first reference the object's rigidbody, set the target the box is supposed to chase and set the force applied to the object's rigidbody. When the method is initialized prefabs used to instantiate game objects are established, the amount of time for object's deletion is reset and the position used to spawn game objects is created. The method SetThrowForce sets the force applied to the object on each update. In the Update method boxes are going to be thrown at the target, force is calculated by adding the target position to target velocity multiplied by the distance between target and the box. When the box collides with its target, it is deleted. This feature is going to be used to teach the agent how to recover from unexpected alterations to its path.

RecordingEnvironmentMaster script will contain scene information and allow access to objects for training and testing purposes. The script will communicate with Recorder.cs to record movement of the agent. After its initialization, recording will be enabled and disabled by pressing the R key. When recording stops, the path is going to be saved to the file. Later, animation index will be defined from user input.

Implementation and execution

User Manual

Project Configuration

The implementation of the project is made in Unity 3D 2018.4.14f1. All scripts produced for implementation purposes were made using C# .Net 4.x. The machine learning functionality is provided by Unity ml-agents.

Download Link

The project implemented as part of this research paper can be downloaded from the following link:

https://github.com/FotisSpinos/Year-4-AI.git

Project Requirements

The project is configured with the following python modules:

ml-agents: 0.15.0.dev0,

ml-agents-envs: 0.15.0.dev0,

Communicator API: API-15-dev0,

TensorFlow: 2.0.1

Python Version

The python version used for operating with ml-agents is: **Python: 3.7.0**. Other versions of Python are supported by ml-agents but there is no guarantee that negative repercussions will not affect the project. The user is able to run the provided trained agents without Python or the mentioned modules. The underlined modules are necessary for training new agents.

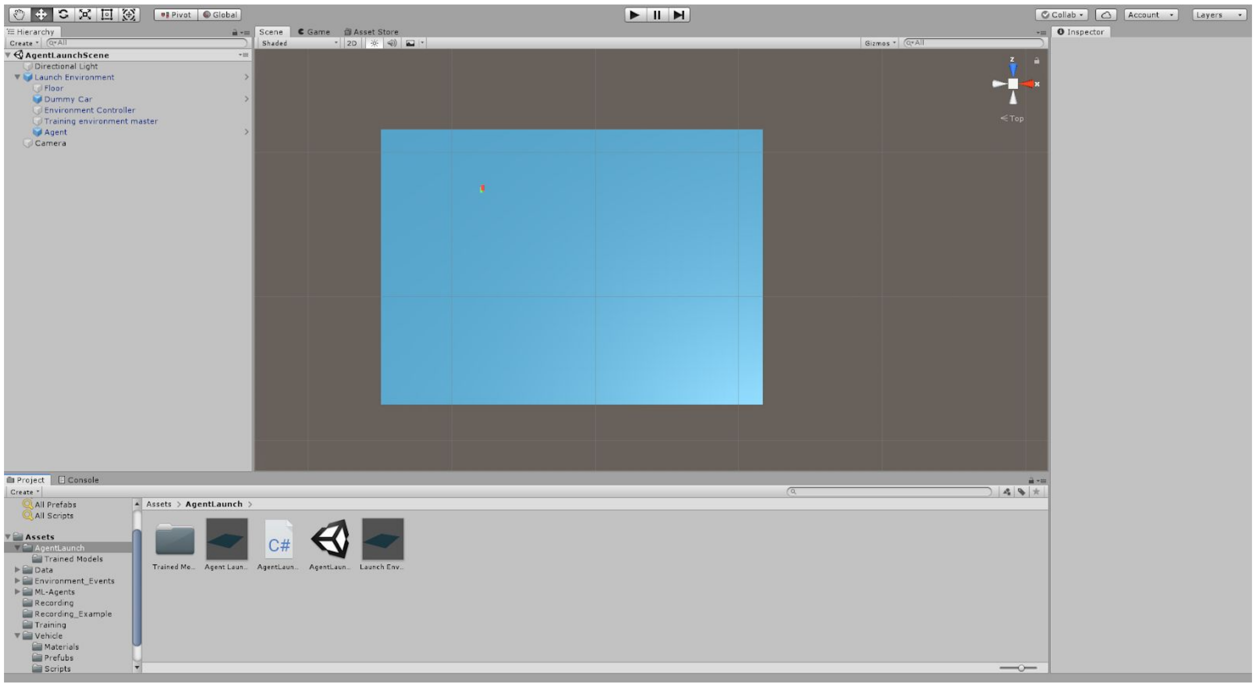
To properly manage ml-agents and the mentioned Python modules we made a Python virtual environment. More information on the supported Python versions along with guidance on creating a Python virtual environment can be found in the installation guide provided by the ml-agents project on GitHub.

The ml-agents project can be found here: https://github.com/Unity-Technologies/ml-agents

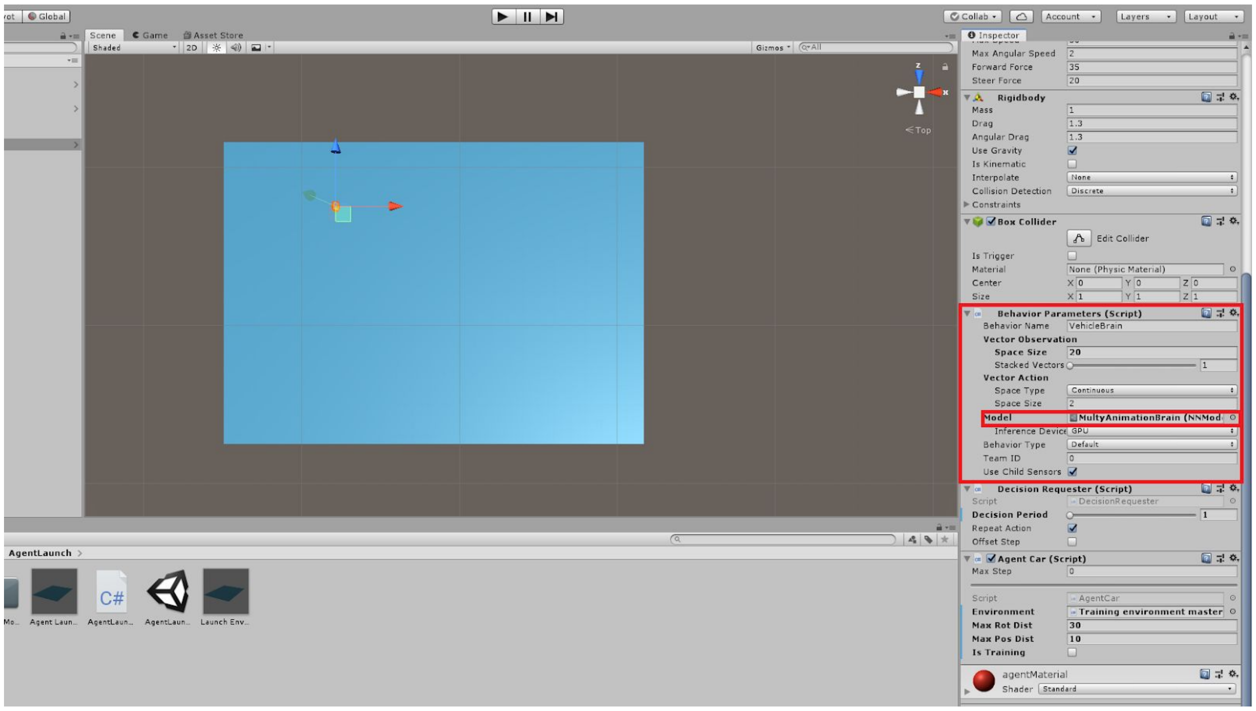
Project Guide

Launching the agent

To launch the agent, make sure that you open the Unity project and navigate on the AgentLaunchScene under the AgentLaunch folder. Once the scene is loaded the editor should look like the following image:



Pressing the play button at the top of the window executes the project and therefore launches the trained agent. To choose a different agent select the Agent game object from the hierarchy found on the left side of the window and set a new model at the Behaviour Parameter component.



```
Max Angular Speed      2
Forward Force          35
Steer Force            20
Rigidbody
  Mass                 1
  Drag                 1.3
  Angular Drag         1.3
  Use Gravity          ✓
  Is Kinematic         ☐
  Interpolate          None
  Collision Detection  Discrete
  Constraints
Box Collider
  Edit Collider
  Is Trigger           ☐
  Material             None (Physic Material)
  Center               X 0   Y 0   Z 0
  Size                 X 1   Y 1   Z 1
Behavior Parameters (Script)
  Behavior Name        VehicleBrain
  Vector Observation
    Space Size         20
    Stacked Vectors    1
  Vector Action
    Space Type         Continuous
    Space Size         2
  Model                MultyAnimationBrain (NNMod
    Inference Device   GPU
  Behavior Type        Default
  Team ID              0
  Use Child Sensors    ✓
Decision Requester (Script)
  Script               DecisionRequester
  Decision Period      1
  Repeat Action        ✓
  Offset Step          ☐
Agent Car (Script)
  Max Step             0
  Script               AgentCar
  Environment          Training environment master
  Max Rot Dist         30
  Max Pos Dist         10
  Is Training          ☐
agentMaterial
  Shader               Standard
```

A folder at the same directory includes several trained objects. Different models can be secreted by clicking on the dot, next to the model variable on the editor or by drag and dropping NN assets from the Project window.

While running the scene, the user can press the 1,2 and 3 keys to execute the following environment actions: throw a box at the agent, apply a random torque vector rotating the object around the y-axis and respawn the agent, by applying an offset to its current position and rotation.

Recording animations

During runtime the dummy car chooses between a set of recorded animations. The project allows the user to record new animations and train new models into mimicking them instead of the one's provided as part of the project. For that purpose the project provides a scene specialized for recording new animations used for launching and training the agent.

To access the recording scene navigate to Assets->Recording->RecordingScene. The scene allows the user to control the vehicle with the WASD keys. During runtime the user can press 1, 2, 3, 4 or 5 to choose an animation slot. Each one of the slots represents the XML file storing the recorded data. To start recording the vehicle's state press the ‘R’ key. Pressing R once again stops the recorder and stores the collected data to the chosen XML file. Take caution, as already stored data in the selected file are being overwritten every time the user stops recording.

Training Environment

The training scene consists of multiple environment objects. Every agent in each environment is used to pass data to the machine learning algorithm. The more environments on the scene the faster the model is trained as the scene explores a greater variety of data. Alternatively or in combination with adding multiple environments, the user can set a higher time scale to increase the time taken for each update.

To start the training process all conditions specified in the project requirements section have to be met. At this stage it would be suitable to check that the Python virtual environment is active. On Windows the environment can be activated by opening cmd, navigating the environment's scripts directory and executing the activate script. If everything is working as intended, the name of the environment will be added before the current working directory. To start training, navigate to the project and execute the following command: mlagents-learn config/trainer_config.yaml --run-id=firstRun. This command required two parameters: the configuration file and the directory where the trained model will be stored. The user can select their own customized configuration file and store the agent to the directory of their choice. After training the user can import the .nn file to Unity.

Developer Manual

Vehicle Controller

We can think of the vehicle controller as the driver of a vehicle. To support classes with multiple behaviours and provide an expandable solution for building custom vehicle controllers we decided to use an interface to represent the vehicle controller. The interface allows us to keep a common set of rules across multiple vehicle controllers and create unique behaviours for each controller in the scene. It also allows us to create complex classes as C# does not provide support for multiple inheritance. Implementing multiple interfaces for a vehicle controller was not used in the project but the approach can prove to be useful for future work.

Classes implementing this interface are expected to override the following methods to provide unique functionality:

void UpdateController();

The UpdateController can be used to define the controller's behaviour.

void InitController(Vehicle vehicle);

The initController(Vehicle vehicle) is used to initialize the object and pass in the vehicle controller being "possessed".

void ResetVehicleController();

The ResetVehicleController method should handle the controller's state when the environment resets.

UserVehicleController & DummyCarController

The UserVehicleController and DummyCarController implement the VehicleController interface. The UserVehicleController is used to control a vehicle object manually through user input. The UpdateController function receives the horizontal and vertical axis defined in the project settings by default. The input captured by the game engine is being applied to the vehicle passed as a parameter. The behaviour resulting from the gathered input translates into a vehicle controlled with the "WASD" keys.

The DummyCarController is used to apply a collection of positions and rotations to the vehicle attribute. The dummy car object uses the data read from the animation files and applies them to the vehicle. The vehicle follows the animations as recorded in the file. The class is also used to retrieve vehicle velocity which is also read from the XML files. The dummy car is not a physics object and retrieving the object's velocity is not possible as the game object's rigid body is inactive. The ResetVehicleController function is used to pick a random animation and a random point from the picked animation. The update function will then continue playing the animation from the selected point. When the animation ends (the script applies the last element of the collection making up the animation) the reset function is called so that a new frame from the recorded animations is played. The collection making up each animation contains vehicle state instances. A vehicle state contains two vector attributes for the object's position and rotation. Unity provides the Vector3 data structure used to contain the game object's position and rotation. These attributes can be accessed with the Transform component attached to each game object. The position and rotation are public variables contained by the Transform class.

During development we identified a bug reducing the quality of our training environment. More specifically, the agent would retrieve an unfair punishment once an animation reaches its end. This happens because the reset function picks and applies the state of a random point from our animation data. This can result in the dummy car "teleporting" to a far location forcing the agent to respawn on the dummy car object. Since the reason the dummy car is being reset is the distance between two objects the agent will retrieve a punishment. More detail on the agent's functionality will be given in the upcoming sections of the document. To avoid the issue of punishing the agent unnecessarily at the end of an animation we used C#'s. The agent subscribes to the OnReset event so that the agent's reset function is called before the distance check is made. This results in the agent being reset without receiving any punishment.

EnvironmentAction

All environment actions implement this interface.

The interface provides the following methods:

void ExecuteAction();

Can be overridden to start / execute the action.

void InitAction();

Can be overridden to initialize the action.

void UpdateAction();

Can be overridden to provide functionality before or after the environment action starts.

Environment Actions: ThrowBoxAction, InvisibleForceAction, TeleportAction

These three classes provide functionality used for the external factors affecting the agent’s state. The throw box action spawns a physics box chasing the agent game object. The invisible force action allows us to directly apply a torque force to the agent at a random direction. Accordingly the teleport action spawns the agent at a nearby location with a random offset from the object’s current position and velocity.

EnvironmentMaster & BaseEnvironmentMaster

EnvironmentMaster is an interface providing the following functions:

void InitEnvironmentMaster();

void UpdateEnvironmentMaster();

DummyCarController GetDummyCarController();

The BaseEnvironmentMaster class implements the EnvironmentMaster interface without providing any functionality. The purpose of the BaseEnvironmentMaster is to provide a blank environment master while allowing us to take advantage of the Unity editor. The editor supports objects inheriting from Monobehaviour. Since BaseEnvironmentMaster inherits from Monobehaviour, classes inheriting from it can be visible to the editor. In addition, the developer can also override the existing functions to provide custom behaviour. This approach allows us to choose the environment master of our choice from the editor, without having to build prefabs or attach the current environment master on each scene or environment.

//provide a picture with the environment master

The purpose of the environment master is to initialize and make all the necessary calls for the purpose of supporting the scene’s requirements. Our program consists of three basic scenes which brings the need for constructing three separate environment masters. Environment masters are particularly responsible for determining when the functionality provided by the initialized objects is executed. For instance, the agent launch environment master checks for user inputs and executes an environment action.

The same can be said when recording vehicle states: When a key is pressed the recorder stores the states observed on each frame. A great advantage of using this approach is that the programmer fully controls the execution order of the program as some objects need to be initialized before others. This is prefered over relying on Unity's execution order as we cannot guarantee which objects are initialized first.

SceneController

An environment should contain a scene controller. The scene controller inherits from Monobehaviour and as a result it exposes a BaseEnvironmentMaster attribute to the editor. This allows us to conveniently assign the EnvironmentMaster of our choice from the editor before runtime. The class contains a Start() and an Update() method. The game engine is responsible for calling these two functions. The Start method is executed once after each execution while Update runs once on each consecutive frame.

RecordingEnvironmentMaster, TrainingEnvironmentMaster, RecordingEnvironmentMaster

These classes inherit from BaseEnvironmentMaster and override the InitEnvironmentMaster and UpdateEnvironmentMaster functions. As a result they provide their own unique functionality to support the needs of each scene.

XmlReadWrite

A singleton class allowing the saving and loading of animations during runtime. The Singleton pattern is ideal since the software needs one instance of this class and access to its load and save functions. To implement this class we used the XmlSerializer class provided by the .NET framework. This made implementation very convenient as the class provides functions for saving and loading objects to and from XML files. As a result we did not have to build a specialised algorithm for these purposes. Therefore the Save and Load methods make use of the Serialize (for storing) and Deserialize (for loading) functions. The object passed to these functions is of type VehicleState tagged as serialisable(which is used to declare that the object can be serialized and deserialized). The VehicleStates contain a VehicleState list. The state of a vehicle is defined by the vehicle’s position, rotation and velocity. These are the three attributes used to describe animation frames.

AgentCar

The AgentCar script represents the behaviour mimicking agent. The implementation of the agent relied heavily on how we used the engine and the mlagents package.

The GameObject containing the AgentCar script should contain a CarVehicle and a Rigidbody component. The Rigidbody enables physics based behaviour while the CarVehicle provides us with functions for controlling the vehicle. To ensure that these two components are attached to the game object while avoiding checking for attached components, we used the Unity tag: RequireComponent. According to Unity’s scripting API documentation, when you add a script which uses RequireComponent to a GameObject, the required component will automatically be added to the GameObject (Unity Technologies, 2020).

Two component scripts provided by the unity package are essential for developing the agent. The game object representing the agent should contain these two components so that necessary configurations are defined before runtime. These two components are the BehaviourParameters and the DecisionRequester script. BehaviourParameters exposes a number of agent attributes to the editor. The developer can assign these attributes before runtime. This allows us to specify the amount of inputs and outputs produced by the agent, the option of using CPU or GPU during training or choosing between training the agent and using a heuristic(to provide default functionality such as manually controlling the agent) which can be used for testing purposes. The DesisionRequester allows to specify the amount of steps required before the machine learning algorithm provides an output.

The ml-agents package provides the Agent class which is primarily used to derive functionality for the purpose of building the agent. The agent makes use of the components described in the previous paragraphs to produce the desired behaviour. The Agent class aids the project by offering required functionality for constructing custom agents. With that being said, the Agent class offers several virtual functions allowing us to provide functionality.

The AgentCar inheriting from Agent and overrides the following functions:

void InitializeAgent()

Called once on each execution, used to initialize the agent object.

void AgentAction(float[] vectorAction)

This function is responsible for translating the vectorAction array into actions followed by the agent, measuring the fitness and providing logic for resetting the environment. This function is executed in each frame during run-time.

The vectorAction parameter is an array representing the outputs retrieved by the machine learning algorithm. The size of the array is defined from the editor. Each number in the array represents an action. The agent class allows us to define the type of output retrieved between discrete and continues.

A discrete action space returns integers. Discrete actions are mostly used for selecting a number of defined actions. The range of values retrieved can be specified in the editor by assigning a value to the Vector Action Space Size variable.

A way of making use of this approach is to define actions for moving and rotating the vehicle. For instance the first input element could represent rotations and the second motion. In our software we could write logic to read these numbers and execute a corresponding action. For example 0 in the first element of the array could suggest that the car turns right while 1 suggests that the car turns left. While this approach is feasible a great drawback is that the software does not take into consideration the speed of the vehicle.

The preferred approach is to use a continuous action which provides us with an array of floats varying between a set max and min boundary. This suggests that floats varying between 1 and -1 can be used to represent a car moving or rotating with a fraction of the car’s maximum velocity. We can think of this as a driver pressing the gas pedal to increase or reduce the vehicle's speed while never exceeding the car’s maximum velocity and acceleration. For example if the method SetVehicleInput retrieves the values 0.5, 0.5 the vehicle will experience half of the max force used for motion and rotation. Therefore achieving half of it’s maximum acceleration. The convenience produced by this approach lies in the fact that we can specify the amount the car moves and rotates rather than just specifying a set of actions defined by a discrete action space.

To ensure that the agent can keep up with the dummy car when it is left behind, the inputs passed to the agent’s vehicle are multiplied with a number above one to allow the agent to exceed the maximum acceleration followed by the dummy car. This is because the car producing the animations would only get values varying between -1 and 1. Therefore the first element of the array represents horizontal motion while the second element represents rotations around the y-axis.

The function calculates the distance between the agent's and dummy car's local position and rotation. The Vector class provided by Unity provides the magnitude attribute which is defined by the square root of: x * x + y * y + z * z. (Unity Technologies, 2020). The class defines two variables: maxPosDist (the maximum difference in position) and rotationDifference(the maximum difference in rotation) to determine if the Euclidean distance recorded exceeds these defined values. If these boundaries are exceeded the scene is being reset by calling the Done() function and the agent receives a high punishment. A punishment can be given to the agent through the SetReward(float reward) function. Both of these functions are defined by the Agent class provided by ml agents.

void CollectObservations(VectorSensor sensor)

This method is used to feed the machine learning algorithm with inputs retrieved during training. Passing data to the agent is achieved by passing data to the sensor parameter. The VecrorSensor provides multiple overloads of the AddObservation function allowing us to pass a range of supported data structures to the agent.

void AgentReset()

Agent reset provides the functionality used to reset the environment. This involves resetting the dummy car by picking a random animation and a random state in that animation. The agent is then spawned on the dummy car object by copying its position and rotation.

float[] Heuristic()

The heuristic provides functionality for allowing the user to control the agent objects. Similarly to the user vehicle controller we used the Input axis retrieved by the engine and passed it as input to the vehicle controller controlling the agent.

Update and AgentUpdate synchronisation issues

Using the EnvironmentController to update the scene appeared to be problematic as the agent and the other objects handled by the TrainingEnvironmentMaster were not updating at the same rate. After some testing, it was noticed that AgentUpdate would be called more frequently than the engine’s update function used by the EnvironmentController. As a result the agent would appear to move significantly faster than the dummy car as the car would update at a higher rate.

To avoid these issues we relied on the agent to make the necessary environment calls at initialization and during each update. That way we could ensure that both the agent and the dummy car would synchronize.

Pilot testing

The goal of the fitness function developed is to make the agent mimic an object with a predefined state. The training scene does not include any external factors altering the agent’s state. Whereas the launch scene allows the user to trigger the provided environment actions.

Each training environment consists of a platform, an agent and a dummy car. Each one of these agents provides data to the model being trained. The research method used was observation. Results are based on observations of the agent behaviour and every test is summarized with information on what has been learned from the test.

Test 01:

The first attempt at testing the agent was used to determine a foundation for the fitness function. This can then be built on by each subsequent iteration. The fitness function used in this iteration is $\underline{F = 1 /(\text{Dummy State - Agent State}})$. This is designed to reward the agent based on the difference (Euclidean distance between the position and rotation recorded by the two objects in the current frame) between the states of the agent and the dummy car. The shorter the distance between the two states is the greater the reward that will be given on the current iteration.

Results:

The fitness function used resulted in the agent maneuvering off course with no ability to recover, and continue to learn the intended path. As the agent does not need to know the limits of the training environment it was possible that the agent would fall off the ground plane making it impossible to recover. It is because the reward function could sometimes go way too high (effectively to infinity) which would result in rewarding the agent far too much for some less important tasks and make it ignore the more important tasks.

Test 02:

In order to solve the problems discovered in the first attempt a new reset function was introduced. The reset function sets the state of the agent to the current state of the dummy car whenever they are too far apart. In order to teach the agent that it should stay close to the dummy car, a punishment will be applied to the agent each time it triggers this reset function.

Results:

The effects of the reset function have been both positive and negative. The positive effects showcased that the agent can now focus on following the dummy car, as it now possesses the ability to learn from its failure. Another positive of the reset function is that it provides a “safety net” for the agent should it fall off the ground plane.

The main negative of the reset function is that the agent learned to exploit it. The agent accomplished this by moving far away from the dummy car in order to constantly gain the highest possible reward provided when the reset function was called. It is likely that the punishment administered when the agent was reset was not severe enough and was counteracted by the total reward gained. In effect, the agent was not using the “correct” method of gaining the highest possible reward, but was still earning points and therefore didn’t behave correctly. It provided some interesting insight on how agents can exploit the reward system in ways of which humans wouldn’t even think of. Reset function proved to be not effective enough to stop the agent from abusing it, that’s why it had to be dealt with in other iterations of the project.

Test 03:

The fitness given to the agent is based on the distance in states formed between the itself and the dummy car. The factors influencing the fitness function is the rotation and position of the two objects. In the previous tests the fitness function was defined by: fitness = 1 / distance. This was problematic as respawning the agent would give an infinite fitness value. (This is because the distance is 0, divisions with zero give infinity). This test made sure to eliminate this issue by checking if the distance is less than 0.1. If it is, the fitness given is a fixed value, otherwise the same function is used (fitness = 1 / distance).

Results:

The agent would still exploit the reset in a strategic manner as it was previously mentioned. Tweaking the fitness value given when the distance is near 0 and the punishment should fix the issue. After increasing the punishment the agent seemed to follow the dummy car's rotations without mimicking it's distance. Resets from the environment would still happen.

Test 04:

The next change will involve increasing the punishment. In addition, the position difference is 80% of the fitness gained while the rotation is the 20%.

Results:

The agent would still exploit the reset function. It has been noticed after the previous two tests that it's not the position difference or the punishment given that required the change. The next possible solution required a tweak in the fitness value giving operation, which would assign high fitness right after reset happens and therefore avoid exploitation of the reward system.

Test 05:

In this test, we introduced a timer setting the fitness value to 0 after a reset occurs. The purpose is to avoid giving the agent high fitness after the reset. When the timer is set to 0 the fitness given is the same as the one described in the previous test. The agent is able to observe a boolean determining if the timer is active (if the fitness value passed is 0 because of the reset).

Results:

The agent could not follow the dummy car (I will keep checking other alternatives for this)

Identified Fix

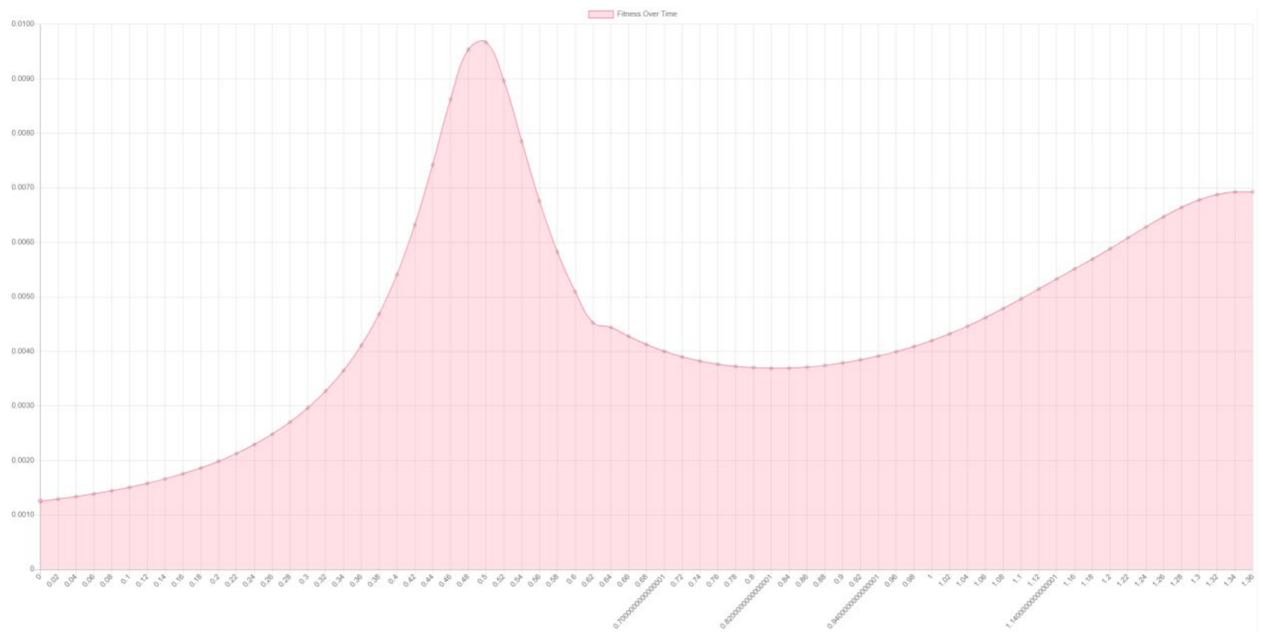
The distance formed by the rotations is problematic for resetting the environment and passing it as input to the neural network. This is because the rotations are expressed as euler angles which vary between 0 and 360. (Distance is therefore inaccurate). An object rotating to the left reduces the value of the angle. After exceeding 0 the angle “resets” from 360.

Test 06:

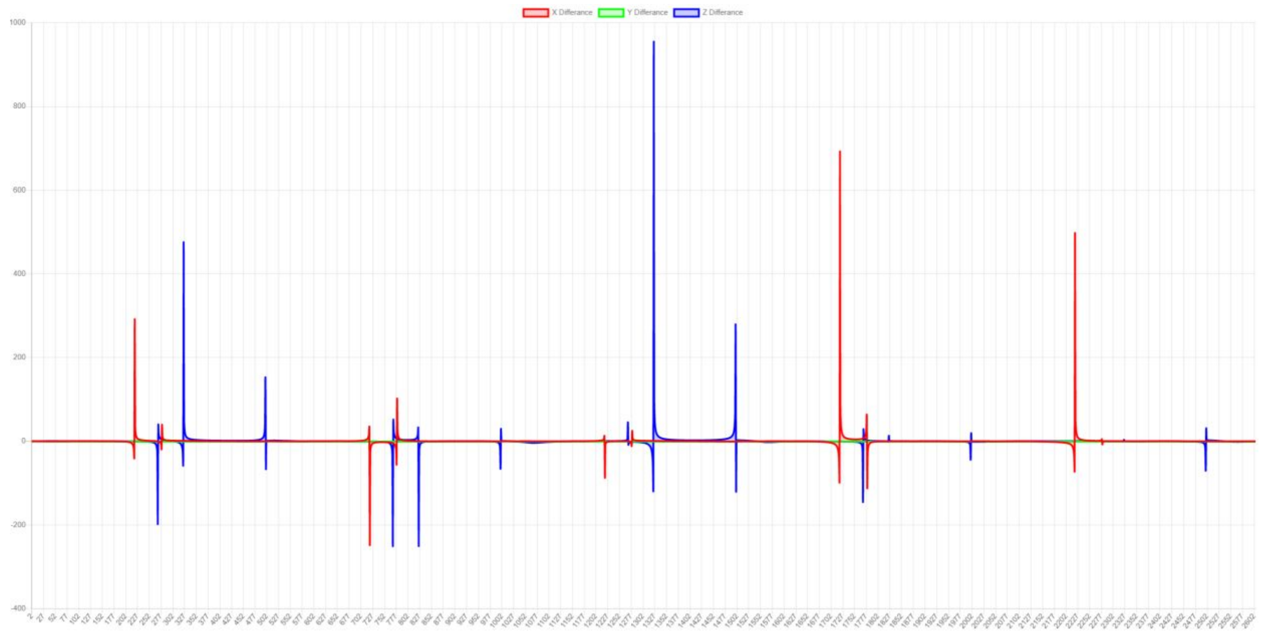
The fitness function calculates the angle formed between the agent’s and dummy car’s forward vector. When the distance surpasses a set value, the environment resets. The most fundamental change is the reset function. The agent is spawned to the dummy car’s position with a random offset applied to its state. The agent’s rotation, position and velocity differs from the observed dummy car’s state. The applied offset allows the agent to move towards the agent without applying a punishment resulting from the distance formed between the two objects.

Results:

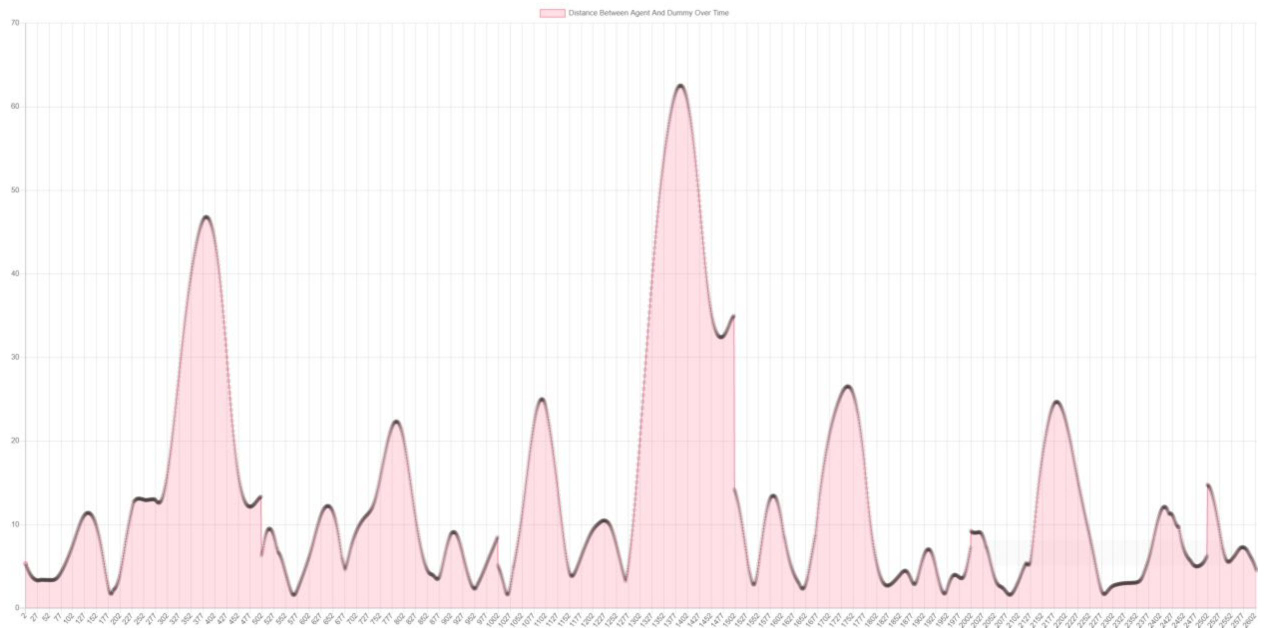
The agent would move closer to the dummy car, outperforming the previous approach in most cases. The fitness retrieved from the agent’s rotation was not as significant as the distance and so the agent ignores the dummy car’s orientation to maintain a close distance. A behaviour would involve the agent moving backwards in turns to sustain a close distance with the dummy car. A disadvantage of this approach is that training times were extremely long. 60 instances of the agent feeding data to the brain would train in an environment running 1000 times faster than normal for about 12 hours. The agent was asked to learn a single animation and still after all this time in training the agent kept improving.



A fitness over time graph captured when the agent is launched



A graph capturing the percentage difference based on how far away the agent is from the dummy car. The data were captured when the agent was launched.



A graph capturing the Euclidean distance between the dummy car vs the time passed. The data were captured when the agent was launched.

Test 07:

This iteration increases the punishment given to the agent when the reset function is called. This increase should ensure that the reward gained when the agent is moved to the dummy car no longer interferes with the punishment administered by the reset function. The agent should now view the reset as something to be avoided.

We learned that the resetting function was not the most effective way of fixing the issue and we needed a more realistic approach, that would allow the agent to learn in a less artificial way. Because of the reset function, the agent would never find a way to get back on a track fully by itself.

Results:

The results of this attempt show improvement to the previous iteration. However, while less frequent the agent will still utilize the reset function in order to get to the dummy car. The agent seems to have been more strategic, only using the reset when the dummy car is making a turn, possibly to minimize the punishment given by the reset.

The less frequent use of the reset has allowed the agent to learn how to better follow the dummy car's position, this hindered the agent’s ability to match the rotation of the dummy car.

Test 08:

In this attempt the agent was rewarded only if the difference in its state and the dummy car's state was not too great. A reward will be given to the agent so long as its state closely resembles the state of the dummy car.

The reset function would work as it did in the previous iteration with it moving the agent back on course but punishing it greatly to discourage its use.

Results:

The agent takes some time before it begins to learn from training. This may be due to a number of factors. One such factor may be the velocity of the agent not being altered by the reset function, making it more difficult for the agent to follow the dummy car once called.

The agent’s speed may be affected by the environment actions influencing its state. Considering that the dummy car may be moving at full speed the agent may be unable to reduce their distance. This causes the agent to be unfairly punished by the reset function.

Test 09:

This iteration corrects the issue where the agent was being punished when the dummy car restarts from the beginning of its path. The function that was used to update the environment was moved to the same update function used by the agent, this fixes the synchronization issues caused by the mlagents code used for the agent. The agent now observes the velocity of the dummy car, as it is a necessary variable for the agent to reproduce in order to successfully copy the dummy car.

Results:

The results of this test were favorable, the agent successfully followed the car with no real issues worthy of note. As this was the case these conditions allowed for the use of multiple environments and a longer training time to better estimate real training conditions.

Test 10:

In this iteration some corrections were made to the observations used by the agent during the training process.

This should provide the agent with a better perception of the current state of the dummy car and allow for more accurate rewards to be received. The rotation vectors used in previous iterations were replaced with the value of the z-axis rotation meaning that the agent and dummy car will now move via the local z-axis values. This will improve the efficacy of the training process as the information of the local x and y axes are unnecessary which may lead to a slower training process if used.

Results:

The results of this test demonstrate the agents improved speed as it follows the dummy car. The agent shows improvements to its recovery after it has been displaced via the environmental interactions.

Test 11:

The current agent has the ability to mimic multiple paths that can be used by the dummy car, however this is still imperfect. The agent seems unable to follow at certain points appearing to lose sight of the dummy car momentarily. The agent displays some improved recovery, however like before it is still imperfect and will be unable to recover if it’s state is altered too greatly.

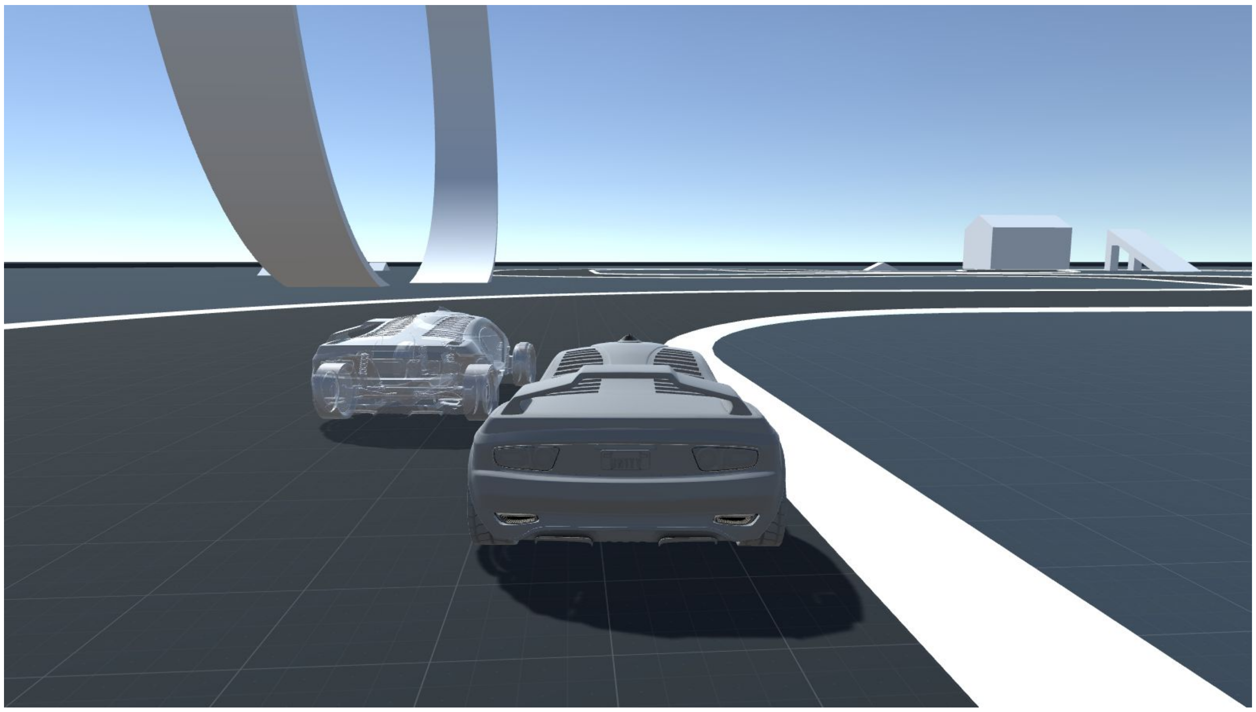
Test 12: Unity Car Controller

For this test, we integrated the car controller offered by the game engine as part of Unity's standard assets. The new recorded animations and agent were based on the unity car. This car features complex geometry and significantly more realistic motion. The challenges provided by this test were not present in previous iterations as the car we developed was influenced by RigidBody attributes and vector forces moving and rotating the object. The velocity of the unity car is defined in relation to the previously recorded velocity and other obtained data like wheel rotations, gear number and applied steering. The training environment provided more complex geometry resulting in the agent moving vertically as the ground was not smooth in places. When recording animations we avoided these points in the map, but some influence was still present in the object's state.

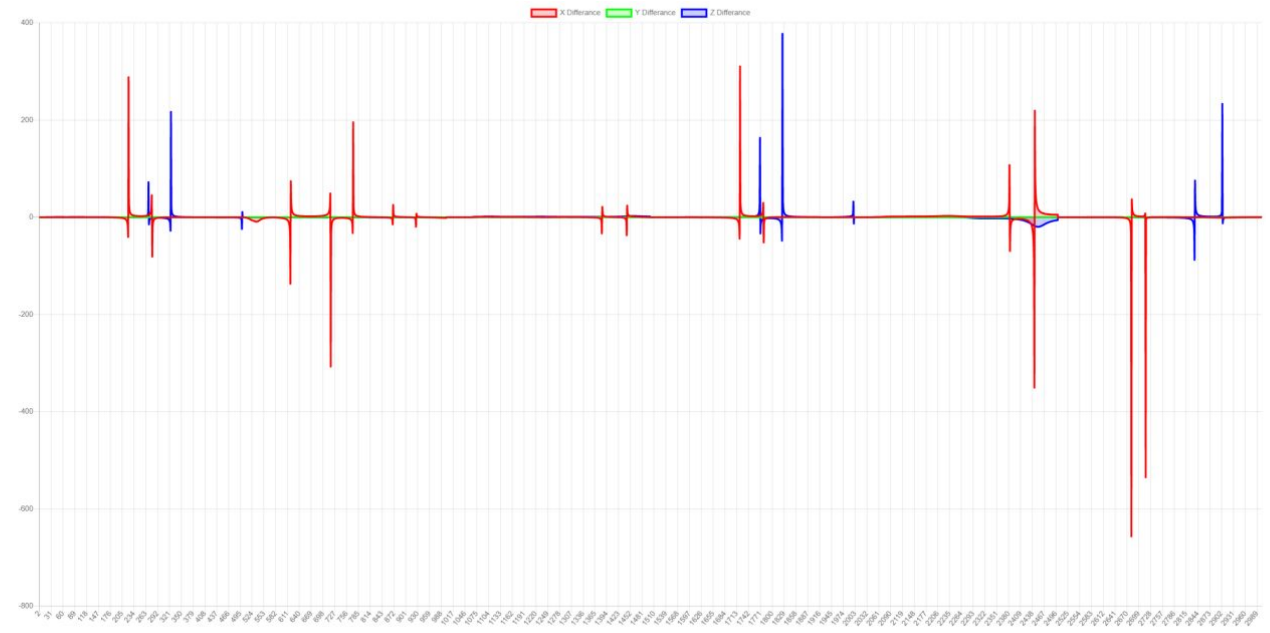
The agent has to be aware of these factors and so the neural network controlling the agent receives significantly more inputs. Because of the introduced complexity the training of the agent was far more demandfull. The fitness function used to train the agent was identical to the agent described in test 11. Using the observations originating from the car controller's state and Rigidbody's attributes used in previous tests was not ideal as the agent showed almost no signs of improvement. Additional research and experimentation contributed in identifying flaws in our approach. These flaws did not influence the other agents greatly as the environment was far simpler.

The first problem identified when launching the trained agent, was that neural network outputs would not vary depending on fed inputs. The values retrieved from the neural network would only reach the set boundaries (In this case 1 or -1). As a result the agent would continually follow one behaviour. This problem was resolved from normalizing the data fed to the network. The involved nodes and links making up the neural networks would have to adapt to extremely high values originating from the retrieved inputs. The process of finding appropriate weight values to lessen the effect of high input values on the network can be very time consuming or could possibly lead to untrained models. The network nodes associated with the input would greatly influence the resulting output, which is not desired as other inputs would not affect the agent's behaviour. This problem was resolved but the agent did not show the desired results. The agent could mimic some parts of the animation but would lose track of the dummy car in some turns.

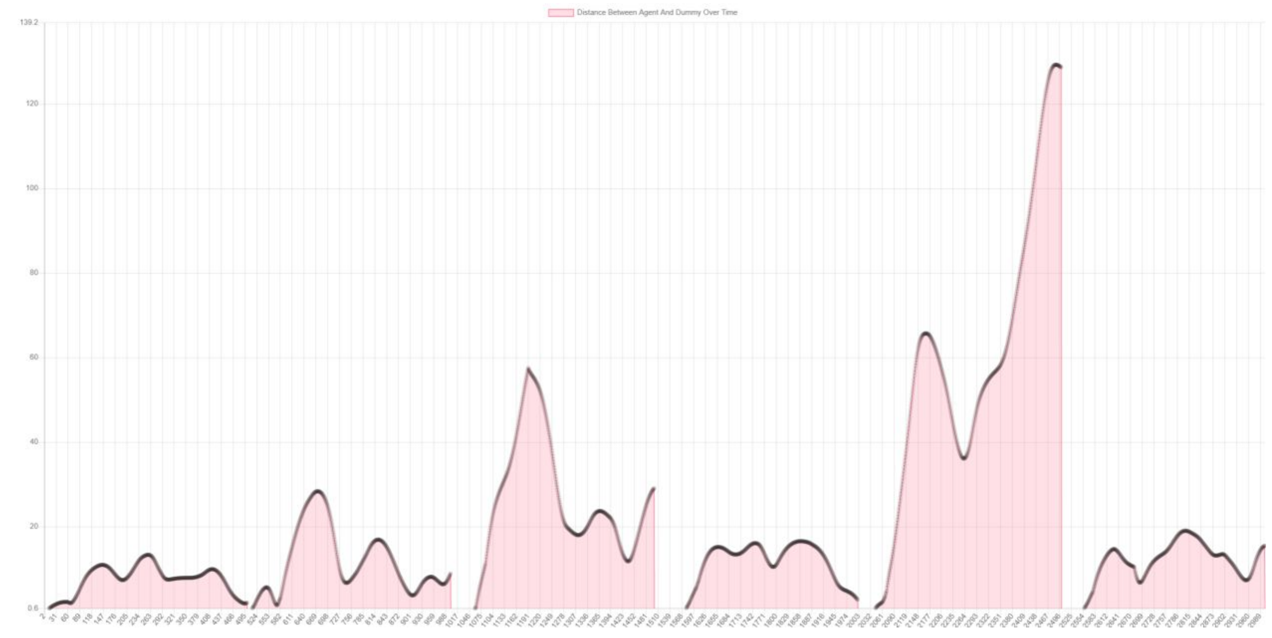
This and previous agents would not be able to keep up with the dummy car on stiff turns. We discovered that there was a flaw in our approach. As mentioned previously, the state passed to the neural network for training consists of the object's velocity, position and rotation. This information can be used by the agent to sense the dummy car’s position since the velocity vector provides both the magnitude of the displacement and direction. There is therefore no information on the agent’s future rotation resulting from torque forces applied to the vehicle’s body. The RigidBody component provides a variable to represent the angular velocity which is used to reset the agent and as data fed to the neural network.



A screenshot of the agent following the dummy car represented by the vehicle with the transparent material



Percentage difference based on how far away the agent is from the dummy values



A graph capturing the Euclidean distance between the dummy car vs the time passed. The data were captured when the agent was launched.

Final results, analysis and conclusions

Evaluation: Scenes

The main program for the project consists of two scenes, the training scene and the launch scene.

The training scene consists of multiple environments, each containing a dummy car following the available paths and an instance of an untrained agent. The purpose of this scene is to teach the agent to mimic the dummy car. The scene performs its intended function well as the use of multiple paths provides the agent with a broader range of training data, minimising the potential risks of over training to a specific path.

The launch scene consists of only one environment containing the dummy car following the available movement paths and a fully trained agent. This scene is used to assess the trained agent by letting it follow a dummy car while its state is being manipulated from external factors. The user can trigger environment events (which are executed on user input) to alter the vehicle's state.The scene performs its intended function of testing the agent well, this is due to the simplicity of the environment.

The use of ml agents was very beneficial to the project. The scripts provided by the package allow seamless communication with the python modules underlined in the user guide. The engine and the package provides a stable environment and ease of development allowing us to conduct multiple tests, gather data retrieved from the objects in the environment and observe agent behaviours.

A potential improvement that could be made to the program would be the inclusion of a GUI for greater useability. The proposed GUI would include: necessary controls such as a dropdown selection box for the user to choose the path followed by the dummy car, basic instructions for the program and on screen messages displaying necessary information regarding confirmation for saving, loading and agent's performance.

Evaluation: Testing

All conducted tests provide information on the agent’s development on each iteration. The tests focus on how well the agent was able to follow the dummy car and its ability to recover if tampered by factors originating from the environment. Each following iteration improves on the agent’s training. This is done through fixing any problems such as bugs or unexpected behavioral decisions. One improvement created as a result of testing was the reset function. The reset function sets the agent’s position and rotation to match the dummy car if it is too far away.

Each iteration of pilot testing consists of observing a trained agent following the dummy car and reacting to any manipulations from the user. Once the test ends any problems are noted and a solution is devised for the next iteration, this process refines the agent each iteration and improves any feature that is not working correctly. One example of this refinement came around after the introduction of the reset function, due to the agent learning to increase its fitness by exploiting the high reward given when the agent is moved to the dummy car's position.

Test 12 was a separate test from the iterative method used in the previous tests.
The purpose of that test was to assess if the logic used in test 11 could be applied in more complex physical bodies. The new car features complex geometry and more realistic motion instead of movement and rotation through manipulation of RigidBody attributes and vector forces. This test highlighted a flaw in our approach. This flaw is regard to the input variables contained within the dummy cars state passed to the neural network for training consisting of the object’s velocity, position and rotation. The agent uses this information in order to locate where the dummy car is, its orientation and its movement speed. This allows the agent to learn the path by following the dummy car. However the information cannot be used to determine the dummy car's potential rotation resulting from torque forces applied to the vehicle’s body. As the agent is unable to determine the new rotation it cannot completely learn how to mimic the dummy car.

Evaluation: Conclusion

The research question this project was aiming to answer, was:

Can an AI agent mimicking the movement path of another entity use deep Q-learning to recover from unexpected interventions diverting it from its learned path?

The agents developed in this project were capable of mimicking the movement path of the dummy car. While following the dummy car, agents could keep up with it the majority of the time, with only minor errors showing up occasionally. The user could manipulate the state of the agent to set it off course, this is used to demonstrate the agents ability to recover. The agent has shown the ability to recover from mild manipulation, however if the agent state is greatly manipulated then it is possible that the agent may not be able to recover.

In order to answer the question, two approaches were used. The first approach was to give the agent a small reward each frame, if its state was similar to the state of the dummy car. However if the agent's position and/or rotation were too different from the dummy car then a high punishment is given instead. To ensure that training is continued effectively, the agent is spawned on the dummy car.

The agent produced is trained quicker than the agent following the second approach. The agent receives rewards for maintaining a short distance in states with the dummy car. The small reward given is not affected by the distance, which results in an agent learning to move and orientate close to the dummy car. The agent produced in this approach is not capable of mimicking the dummy car's state in great accuracy as the agent's priority is to avoid the high punishment given when the environment resets.

The environment uses a fixed boundary (the maximum allowed distance in terms of position and rotation formed between the agent and the dummy car) to determine if a reset resulting in negative reinforcement occurs. Reducing the values of this boundary will not allow the agent to move away from the dummy car. The task undertaken by the agent is more difficult as the agent has to find a way to stay nearer to the fixed behaviour object. Training this agent would therefore take significantly longer, but additional research on this approach could produce a more robust agent.

The aim of the second agent was to improve on the accuracy produced by the previous approach in mimicking the dummy car. The fitness function gives a reward that is inversely proportional to the distance in states. The closer the agent matches the dummy car the higher the reward that is given. Should the distance between the agent and the dummy car become too great, the reset function will be called in order to move and rotate the agent to a random point within the set boundary. If the reset function is called then the agent will receive a high punishment.

The agent produced showed the ability to mimic the agent’s position in a greater accuracy than what was displayed by the previous agent. It was observed that this agent would not prioritise the dummy car’s rotation. On stiff turns, the agent would choose to partially turn and move backwards while admittedly maintaining a close position with the dummy car.

As mentioned in the test pilot, the distance in the positions is the main factor influencing the fitness. While rotations had an effect, they were not as significant. In addition, the values were not normalized and so distance values could far exceed rotation values making their contribution to the resulting fitness value even less significant. While this approach increases accuracy in positions, the agent requires longer training sessions before a satisfying result is achieved.

The last test shows that both approaches presented some issues. For instance normalizing data and feeding the angular velocity as input to the neural network could produce better results as the agent can more accurately predict the future state of the dummy car. As data fed to the neural network were not normalized, the outputs retrieved would be equal to 1 or -1. In addition to outputs being equivalent to the maximum and minimum boundaries the process of identifying appropriate weight values can greatly increase training times before the agent produces an improved behaviour.

Overall issues with the second approach would include small things such as the agents difficulty with following the path on stiff turns. A larger issue would be that the agent does not go as near as possible to the dummy car, as the fitness function is primarily influenced by environment resets which result in high punishments. Due to the small reward given each frame that a reset does not occur, the agent’s reward is not directly affected by its distance with the dummy car in. Instead the agent is rewarded by keeping the difference in their states within the set boundary. The videos provided in the next section show that the agent does not choose to minimise the difference in their states but instead avoid resets.

Some promising results were produced in this research and additional experimentation could lead to agents recovering from more impactful alterations. The behaviour mimicking fitness function could be modified to increase the agent’s ability to mimic the observed behaviour. A number of additional experiments could help assess the behaviour produced by the developed agent. The agent was not assessed on its ability to follow a dummy car controlled by a user or paths not used in training, thus these areas may be a good start for further development.

Additional research and development may produce a robust agent mimicking the fixed behaviour in complex environments. Areas such as games could benefit from this type of robust agent, as interactions between the agent with a player will not require the developer to explicitly write any additional instructions once the training ruleset has been created.

The field of robotics may also benefit from AI that can recover from unobserved interventions. As it is unreasonable to overuse sensors on a robot, as well as being inefficient in terms of data handling and usage of physical space. Robotic AI with the ability to recover would be useful in scenarios such as, recovering from being displaced from its movement path. A real world example of this displacement could be a mild collision with another entity (e.g robot or human) resulting in the robot being moved off course.

Video clip

https://drive.google.com/file/d/1hxcQhZ06i3hKmhw_6idc6FmpYy2hxHDl/view?usp=sharing

https://github.com/FotisSpinos/Year-4-AI/tree/master/Recordings

References

Technologies, U., 2020. *Unity - Scripting API:*. [online] Docs.unity3d.com. Available at: https://docs.unity3d.com/ScriptReference/index.html [Accessed 10 April 2020].

Kevin, B., Simon, C., Daniel, H., James R. F. Ubisoft Montréal. 2019. *Drecon: Data-Driven Responsive Control Of Physics-Based Characters - Ubisoft Montréal*. [online] Available at: https://montreal.ubisoft.com/en/drecon-data-driven-responsive-control-of-physics-based-characters/?fbclid=IwAR0NfrORnMs2CdWmHSVcUvb7gVonYI6BMyJGLEvMtWKr2cG0dXS_mZkY8tU [Accessed 10 April 2020].

ergaMin, K., Clavet, S., Holden, D. and Forbes, J., 2020. *Drecon: Data-Driven Responsive Control Of Physics-Based Characters*. Available at: https://dl.acm.org/doi/pdf/10.1145/3355089.3356536

freeCodeCamp.org. 2018. *An Introduction To Q-Learning: Reinforcement Learning*. [online] Available at: https://www.freecodecamp.org/news/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc/ [Accessed 1 April 2020].

Fuks, L., Awad, N., Hutter, F. and Lindauer, M., 2020. *An Evolution Strategy With Progressive Episode Lengths For Playing Games*. [online] Ijcai.org. Available at: https://www.ijcai.org/Proceedings/2019/0172.pdf [Accessed 10 April 2020].