INTEGRATION OF MACHINE LEARNING MODELS IN A 3D SIMULATED ENVIRONMENT FOR SURFACE SHIP NAVIGATION

MASTER’S THESIS IN COMPUTER ENGINEERING

JUHA AARNIO 1901630

Supervisors: Dr. Sébastien Lafond and Dr. Sepinoud Azimi
Åbo Akademi University
Faculty of Science and Engineering
2021

Abstract

This master’s thesis describes the process of integrating data generated by machine learning models into a simulation developed using AILiveSim, a tool running on Unreal Engine. The goal is to use this data to improve the accuracy and performance of the simulation. To successfully achieve these goals, knowledge of both machine learning and Python is required, but this thesis focuses more on the integration of the machine learning models into AILiveSim rather than the development and optimization of the used machine learning models. This is done by reading relevant journals and scientific articles from different technology and science -oriented databases. The goal of this research phase is not only to get a better understanding of the technologies used in the project, but also to get a better overview of how the technologies are used in today’s industry. The gained knowledge during this phase is used to plan the implementation and integration of the machine learning data using best industry practices. After the research phase the project proceeds to the implementation phase, where the work begins by evaluating and examining the used machine learning algorithms and data to choose the best approach for integration into the simulation. This implementation phase also consists of testing and analysis to evaluate how well the expected benefit of the integration manifests in the actual simulation. Finally, at the end of the project the results are synthetized and the effectiveness of the integration in improving the simulation was assessed.

Contents

1. BACKGROUND OF THE PROJECT

Simulations have been used extensively in the maritime industry to both train and educate seafarers and to test out new systems and equipment in simulated environments. Many maritime agencies and companies own or rent access to simulation systems to either train their personnel or try out the behaviour of sensors and other equipment without having to field the system in a real world, which saves costs and labour. One such system is AILiveSim, which is used to simulate maritime vessels in realistic environments and simulate the performance of different sensor setups and systems. Simulated sensors in AILiveSim generate data which can be used to make decisions within the simulation and to analyse the performance of the vessels in different scenarios. In this thesis, methods to apply machine learning algorithms are explored to augment the simulation in a project using AILiveSim.

More specifically, this thesis will focus on integrating pre-existing reinforcement learning frameworks and use the network to teach a vessel in the simulation to perform a given task, which in the context of this thesis is collision avoidance.

Machine learning integration offers several benefits to simulation, where it can be used to process vast amounts of data with minimal need for human intervention or to analyse outcomes of different behaviours and actions. The collision avoidance implementation is intended as a proof of concept for reinforcement learning integration into AILiveSim, which can be used as a future reference when implementing more complex machine learning functionality. Keras-RL was used as the reinforcement learning library and Open AI Gym was used as the environment in the project. AILiveSim does not have inbuilt integration with Open AI Gym or Keras-RL and there is very little existing work on achieving integration between often used reinforcement learning libraries and AILiveSim, the steps to create interaction between AILiveSim and Open AI Gym are outlined in this thesis.

1.1 AILiveSim

AILiveSim is a tool built on top of Unreal Engine. The tool is used for simulating autonomous vehicles and vessels in realistic situations, allowing testing and prototyping of different autonomous solutions for vessels and vehicles and to prototype the effects of different sensors setups. AILiveSim allows the simulation of sensors and sensor data, which can be used to alter the behaviour of the simulated vessels.

The functionality in the tool is implemented using Python scripts which are used to interact with the vehicles and sensors, and process the data generated by the sensors. To achieve this, several Python libraries like TCPClient and ALSClient are used to connect to the control sockets of the different vessels and sensors. The simulated vessels reside in levels called scenarios which in turn consist of one or more situations. A situation is a set of vessels and other objects in certain arrangement, which can then be reused in different scenario. For example, a situation could be three ships moving around each other. This situation can then be used in different scenarios, for example, on a harbour scenario and then on an archipelago level. One scenario can contain multiple situations and subscenes. The scenarios are defined as XML files, in which the desired situations and subscenes are added by using XML tags. These XML files in turn can then be loaded using Python scripts to load the scenario and all the included situations. (Leudet et al. 2019)

1.2 Machine Learning and Artificial Intelligence

Machine learning is a branch of artificial intelligence which focuses on using structures called neural networks to process and refine data. Neural networks are a set of nodes arranged in layers, which represent mathematical functions. Using machine learning, the system can adapt itself and make predictions based on earlier, trained data. Common use-case for machine learning is image processing, which allows the system to detect and recognize images and object from videos and images. Other ways to leverage machine learning is to refine massive amounts of data to get statistically significant results and viewpoints. In general, machine learning allows systems to adapt to changing environmental situations and events.

For example, many retailers employ machine learning algorithms to predict needed stock levels for certain products or even buying habits of certain customer groups. Predictive models attempt to guess future events and trends based on existing data whereas descriptive models are used to get useful insights from the data. (Alpaydin, Bach, 2014.)

1.3 Simulation

Simulation is a system that imitates real-life interactions and events in a controlled environment. These simulations can be used to examine and train for real-world interactions and operations without risking actual equipment or human lives. Simulations rely on accurate modelling of the examined phenomenon or situation to effectively recreate the desired situation and to effectively prepare or train the people undergoing the simulation.

An example of such simulation is simulation training used in training nurses and other healthcare workers. In such a simulation, a medical procedure is performed on an anatomical doll to get experience with the procedure without the risk of causing injury to a real patient. (Sandford, 2010.) A subset of simulations are computer simulations, which are run on computers to simulate the desired phenomena. These simulations are performed in a virtual environment, where the developers and scientists have crafted a simulated representation of necessary physical laws and effects. Computer simulations allow simulation of things that are not possible, or at least extremely difficult, to simulate using conventional experimental setups. Examples of such simulations are systems modelling the behaviour of different physical phenomena, for example fluid and molecular dynamics. These systems are used in research and scientific endeavours to predict the behaviour of complex systems to get useful insights without having to set up expensive experiments in laboratories.

While these simulations cannot completely replace lab experiments, they are useful in complementing the research process by allowing the researchers to better select the best experiments for actual lab experiments. Virtual simulations can also represent traditional environments and situations in the virtual world. For example, the simulation performed by nursing students can be performed with virtual reality equipment in a virtual environment. These virtual reality simulations allow presentation of aspects that might be very expensive with real equipment. Virtual reality simulations allow the subjects to experiment and train with virtual representations of expensive equipment without risk of personnel or material damage. Examples of such simulators are simulated learning environments focused on maintenance of ship engines or factory equipment.

(Sheldon, 2012.) Using machine learning and simulations in tandem is not an entirely new and unresearched topic, there exists earlier implementations and applications using such integration to achieve better results. An example of such system is the use of machine learning in petroleum engineering to evaluate quality of reservoirs of oil. To evaluate the quality of a reservoir, various imaging methods are used to get imagery from the reservoir pores which house the oil.

The different imaging methods all have their strengths and weaknesses, but the important challenge is to get enough detail in a 3D image. The most detailed methods can only generate 2D images, but for proper modelling, 3D images are required. High detail in a 3D image can be achieved by using a machine learning algorithm to upscale the low-detail 3D image to be more suitable for reservoir analysis.

Another way is to use the 2D image as a basis for modelling a 3D pore structure using machine learning. This more accurate data can then be used as a basis for simulation to get data that better reflects the real-world reservoir than without the machine learning enhanced data. (Dean S, Yan, 2020.) In this example, the simulation is enhanced by feeding it data that has been refined with machine learning outside of the simulation. Aforementioned approach reflects quite well the initial status of the thesis project involving AILiveSim project where existing system is augmented with integration of machine learning, but there is also a possibility to use machine learning directly within the simulator to make it more realistic. For example, machine learning could be used to make the components of the simulation to react to unexpected events in real time and learn new, unique behaviours.

1.4 Description of the project

The goal of this project was to implement a way to integrate reinforcement learning algorithms into the AILiveSim tool and use this integrated reinforcement learning solution in a productive way to create functionality that would not otherwise be possible or easily implementable in AILiveSim.

The project's focus was on the integration itself and with the suitability of the chosen technology stack. The project also consisted of developing a test case which could be used to test that the integration was working as intended. The integration phase was also used as an opportunity to document the steps to achieve integration of the reinforcement learning technology.

The thesis starts with a cursory overview of the central concepts relevant to the project implementation, while also exploring the theory and use-cases behind the concepts. This was done to get a better overview of the possible use-cases where the developed solution could be used productively in practical settings, which could play a role when choosing the technology stack for the implementation.

The following chapter will go over the AILiveSim tool, presenting core functionality necessary for the integration project. Commonly used features and concepts are also explained to give a better overview of the AILiveSim and its intended use-case, to give some context on how the reinforcement learning fits in. Third chapter goes over the chosen technology for the integration, explaining the basics of Keras-rl and expanding on the central theoretical concepts important for the integration process and for fine-tuning of the solution. Fourth chapter goes over the actual integration process, explaining the code structure, the chosen approach and going over the proof-of-concept scenarios and their implementation. The theory from the third chapter was applied here to fine-tune the parameters and setup of the scenarios. The fifth chapter goes over the results of the integration approach and also discusses some alternate use-cases as well as limitations and shortcomings of the achieved implementation. Fifth chapter also includes some general discussion about the solution.

The project also included development of four workshops intended for training sessions with the tool. Three of the workshops dealt with concepts central to AILiveSim usage, going over pathfinding, overrides and sensors. Fourth workshop was produced as a result of the thesis project, using the lessons learned during the project to produce training material for training people to integrate machine learning algorithms into AILiveSim and employ it productively.

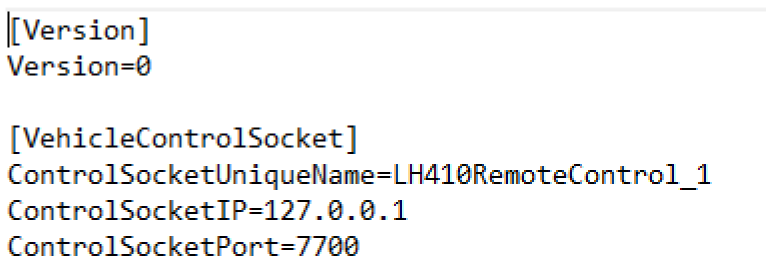
2. AILIVESIM OVERVIEW

AILiveSim is a 3D simulation tool, which can be used to simulate various situations involving autonomous vehicles and vessels. AILiveSim can be used to simulate and test vehicles using different sensor setups and behaviours implemented using Python scripts.

A typical AILiveSim setup contains a scenario and one or more situations, representing different environments and settings where the simulated situation happens. For the user of AILiveSim the most relevant parts are the editors used to create scenarios and situations. These editors are accessed from a menu inside the tool and are for the most part used to create the environments and setups that are to be used for the simulation.

A typical workflow for AILiveSim starts with a creation of a scenario using the scenario editor. After creating and loading the scenario, next step is to either create a new situation or to assign an existing situation to the scenario. To implement the actual functionality for controlling the vessels and operating the simulation itself, python script files are used. The same script files are also used to specify which scenario to load. Python scripts can be used to implement features like repeating the scenario with different parameters, while using overrides to control two vessels on collision course to test out collision avoidance algorithms.

Interaction and controlling of objects in AILiveSim scenarios and situations are achieved with TCPClient library and connecting to the defined control socket of the object. All sensors and controllable objects have a control socket consisting of a network port and IP address. The IP address and port can be configured by the user by modifying a configuration file belonging to the object controlled by the socket.

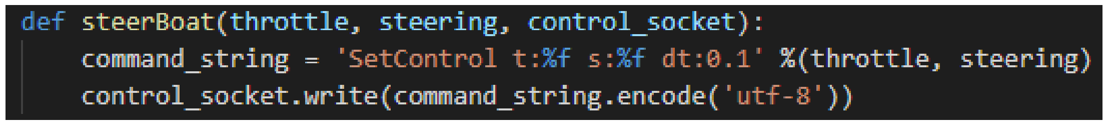


```
[Version]
Version=0

[VehicleControlSocket]
ControlSocketUniqueName=LH410RemoteControl_1
ControlSocketIP=127.0.0.1
ControlSocketPort=7700
```

Figure 1. Example of a control socket configuration

Configuration file in Figure 1 defines a control socket for an ego boat, which is an object that can be either controlled directly or by sending commands remotely using command strings.

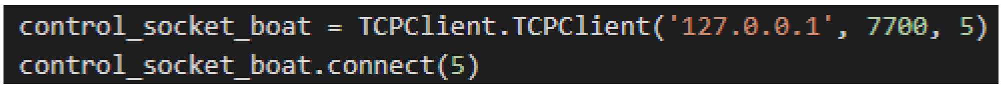
Command strings are used to send commands through control sockets to assume control of objects. The most typical case is sending control commands to an ego boat to set steering and throttle. The ego boat is controlled by sending a control string containing values for steering and throttle through a command socket. The command string is simply an UTF-8 encoded string of characters formatted according to the AILiveSim API guidelines.



```python
def steerBoat(throttle, steering, control_socket):
    command_string = 'SetControl t:%f s:%f dt:0.1' %(throttle, steering)
    control_socket.write(command_string.encode('utf-8'))
```

Figure 2. Example of using command string to control a boat.

Values for throttle and steering are preceded with SetControl -operator. Values for t and s stand for throttle and steering, respectively. After the command string has been constructed, it is written into the control socket using TCPClient's write -function. Dt stands for deltatime operator, which can be used to limit execution time of commands. This allows limiting duration of commands in cases where it is not desirable to have overlap on consecutive commands, for example, when aligning command duration with timestep duration in the case of reinforcement learning approach or other system relying on timesteps. Before control socket can be written into, a connection must be established to it. A control socket must also be defined before it can be used.

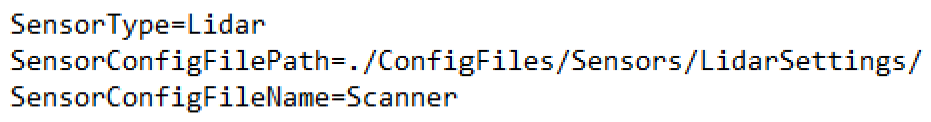


```
control_socket_boat = TCPClient.TCPClient('127.0.0.1', 7700, 5)
control_socket_boat.connect(5)
```

Figure 3. Defining a socket for controlling a boat and connecting to it.

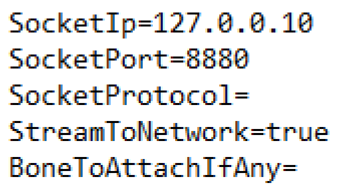
Control socket is defined as TCP connection which is established using TCPClient library. The IP address and port must correspond with the IP address and port defined in the object's configuration file for the connection to succeed. Sensors also work in a similar way, where the data gathered by the sensors is read from socket defined in sensor setup file.

Sensor sockets behave like control sockets, having an IP address and a port, and connected to using TCPClient. Sensors are attached to vessels using the sensor editor, from where a base to place the sensors is chosen first, after which the desired sensors are picked from a menu and then attached to the base vessel. After saving the sensor setup, a setup file is created for the sensor setup, where the sockets used to access different sensors are defined.



```
SensorType=Lidar
SensorConfigFilePath=./ConfigFiles/Sensors/LidarSettings/
SensorConfigFileName=Scanner
```

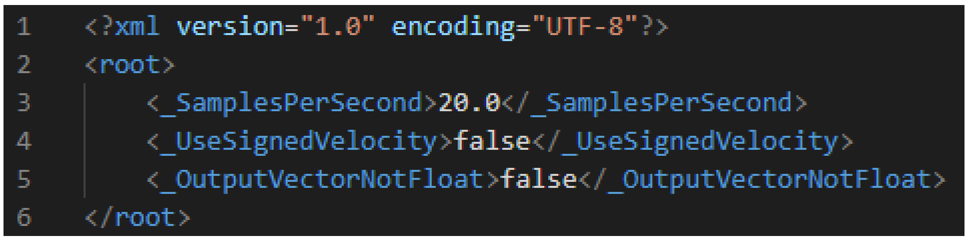
Figure 4. Example of a setup file entry for a lidar sensor



```
SocketIp=127.0.0.10
SocketPort=8880
SocketProtocol=
StreamToNetwork=true
BoneToAttachIfAny=
```

Figure 5. Socket settings for a lidar sensor

In addition to the setup file, each sensor type also has its own configuration files to modify the behaviour of different types of sensors. As an example, these configurations can be used to alter the format of data returned by a speedometer sensor.



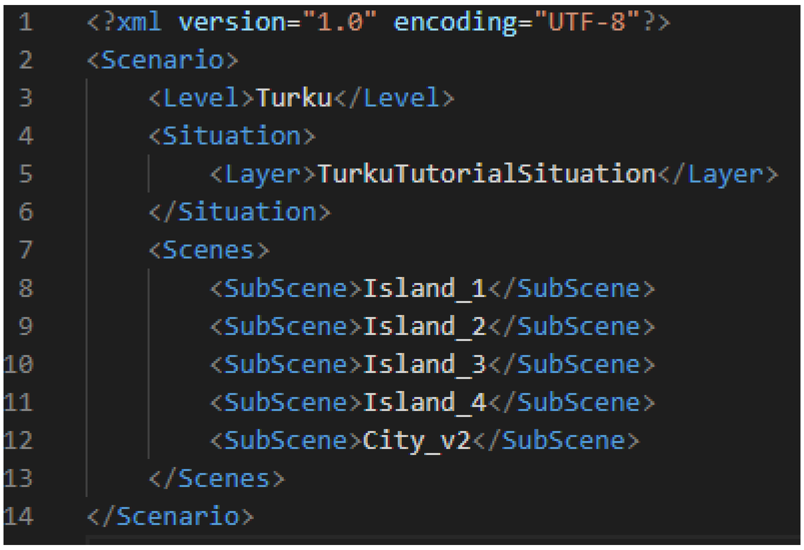
```xml
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <_SamplesPerSecond>20.0</_SamplesPerSecond>
    <_UseSignedVelocity>false</_UseSignedVelocity>
    <_OutputVectorNotFloat>false</_OutputVectorNotFloat>
</root>
```

Figure 6. Speedometer sensor's configuration file.

The configuration file for a speedometer (Figure 6) allows changing of sample rate and format of the returned data.

2.1 Scenarios and situations

A scenario in AILiveSim is represented by an XML file containing information like situations loaded with the scenario and level and possible subscenes associated with the scenario. A single scenario can have multiple situations associated with it and one situation can be used in multiple scenarios.



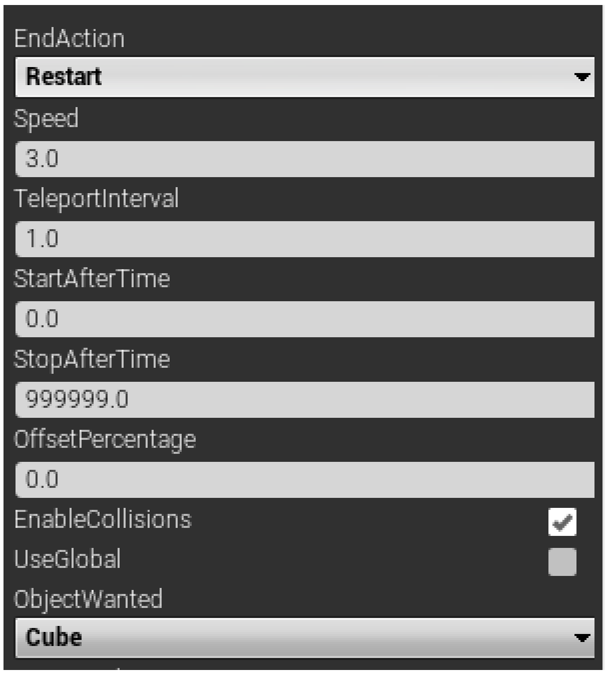
```xml
<?xml version="1.0" encoding="UTF-8"?>
<Scenario>
    <Level>Turku</Level>
    <Situation>
        <Layer>TurkuTutorialSituation</Layer>
    </Situation>
    <Scenes>
        <SubScene>Island_1</SubScene>
        <SubScene>Island_2</SubScene>
        <SubScene>Island_3</SubScene>
        <SubScene>Island_4</SubScene>
        <SubScene>City_v2</SubScene>
    </Scenes>
</Scenario>
```

Figure 7. Example of a scenario XML file which loads a level located in Turku and the situation TurkuTutorialSituation.

While scenarios define the levels and subscenes, a situation usually contains the dynamic components of the scene, like vessels and obstacles like buoys. These objects can then be manipulated with Python scripts by assigning them control commands or by using splines. Splines are situation components which can be used to apply movement to different components in the situation.

A spline is essentially a line drawn between user-defined points. Any object nearby to the spline will try to follow the spline from a distance. The distance the vessel will try to follow the spline from is random, where the upper range of the value can be defined by the user modifying the spline's channel width value.

AILiveSIm also has an animation spline object, which works similarly to the normal spline, with some differences. An animation spline is not followed by any nearby object, but rather, the object on the spline can be chosen from a dropdown menu from the spline's configuration menu (Figure 8).



EndAction	Restart
Speed	3.0
TeleportInterval	1.0
StartAfterTime	0.0
StopAfterTime	999999.0
OffsetPercentage	0.0
EnableCollisions	✓
UseGlobal	
ObjectWanted	Cube

Figure 8. A set of options offered by animation spline configuration menu.

The relationship between a scenario and a situation can be represented as a hierarchy, where a scenario contains one or more situations, and the scenario defines on what level the situations are taking place (Figure 5).

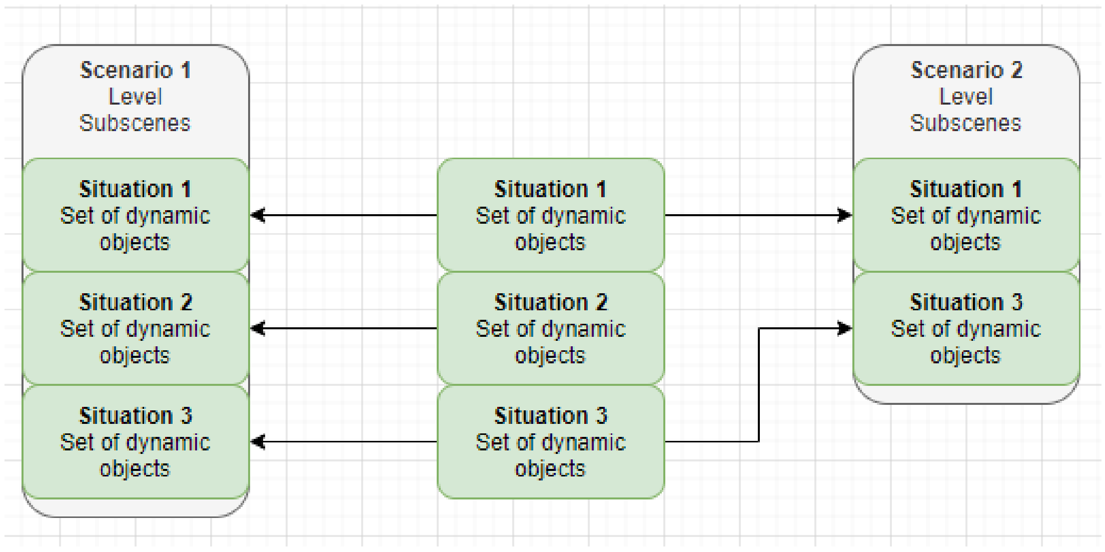


Figure 9. Example graph of the relationship between situations and scenarios.

Situation features like object positions and names are represented by configuration files created by AILiveSim upon saving a situation. These situation files can be read to extract data like object positions and count of objects.

Through these config files, the user can also define aliases to different objects to make referencing them easier when used in scripts. For example, a buoy object might have an object ID BP_BuoyPlacer_C_2147477185, which would have to be used whenever that object is referenced in code.

By assigning an alias, for example Buoy01, the object can be referenced by much easier to remember name in the code by using the alias instead of the full ID.

```
Alias=Waypoint1
InstanceName=BP_BuoyPlacer_C_2147481927
ClassName=BP_BuoyPlacer_C
```

Figure 10. Assigning an alias to a buoy object using situation's configuration file

An object can be given alias by adding Alias -entry before the object's InstanceName -entry.

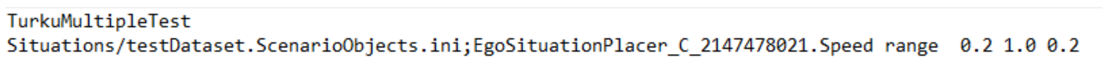
2.2 Overrides

Overrides are a set of variables that can be altered without touching the code.

Overrides are useful for repeated tests, where it is desirable to execute the same situation multiple times with different values for speed or any other variables and see how the changes affect the outcome.

For example, the user can define a scenario where there are two boats on collision course, and the test is repeated five times with each repeat test using a different speed for one of the boats. Overrides are defined using an override definition file, which is a text file containing the required information.

To define an override, the file must contain the variable which the user desires to alter between test runs, the starting value of the variable, the maximum value of the variable, and the stepping defining the amount the variable is changed between test runs. AILiveSim automatically generates a test set based on the defined overrides.



```
TurkuMultipleTest
Situations/testDataset.ScenarioObjects.ini;EgoSituationPlacer_C_2147478021.Speed range  0.2 1.0 0.2
```

Figure 11. Example of an override definition altering speed of an ego boat object.

For example, in Figure 11, override is used to modify the speed of an ego boat between test runs. The speed starts from 0.2, maxes out at 1.0 and is incremented by 0.2 between test runs. AILiveSim will automatically generate the correct number of test runs based on the override definition. In this example, five simulation runs would be generated and ran, where each iteration increments the speed of the boat by 0.2.

2.3 Folder structure of AILiveSim

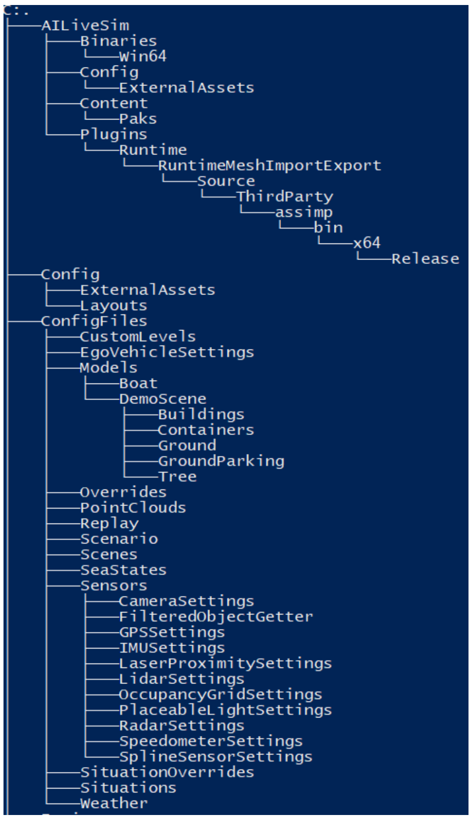
AILiveSim components and config files mentioned above are located in different folders within the AILiveSim main folder. For the purposes of this project, only the folders containing the python code, configuration files, and scenario and situation files are described.

These folders are the Config, ConfigFiles and Python -folders, with rest of the folders mainly storing core components of AILiveSim. The code used to control

AILiveSim simulation and add behaviours to the objects and vessels are in the Python folder, with this also being the folder where the code for the machine learning was placed. Config contains configuration files for general features of the simulation, used to control things like default objects available in the in-tool editors and default file locations for sensor configuration files. This folder was rarely needed during the project, with the config files largely being more suited to a more customized approach for simulating a particular scene.

The AILiveSim documentation should be consulted for more detailed explanations of the different settings. ConfigFiles -folder contains several subfolders containing configuration files for individual objects and features used in AILiveSim, including configuration files for the Ego boat -object and the different sensors. The Scenario subfolder contains the xml- files for setting up and configuring the scenarios.

The Situations folder, respectively, contains configuration files for the situations. EgoVehicleSettings contains the files to configure control sockets for ego boat and also sensor setup files for configuring positions and rotations of installed sensors, among other things. Remaining folders in the ConfigFiles contain configuration files for many aspects of AILiveSim, ibcluding sensors and weather. The licence folder in the main folder contains the licence file, which must be downloaded separately after installing AILiveSim. The licence file must always reside in the folder or AILiveSim will not start.



```
C:.
├───AILiveSim
│   ├───Binaries
│   │   └───Win64
│   ├───Config
│   │   └───ExternalAssets
│   ├───Content
│   │   └───Paks
│   └───Plugins
│       └───Runtime
│           └───RuntimeMeshImportExport
│               └───Source
│                   └───ThirdParty
│                       └───assimp
│                           └───bin
│                               └───x64
│                                   └───Release
├───Config
│   ├───ExternalAssets
│   └───Layouts
├───ConfigFiles
│   ├───CustomLevels
│   ├───EgoVehicleSettings
│   ├───Models
│   │   ├───Boat
│   │   └───DemoScene
│   │       ├───Buildings
│   │       ├───Containers
│   │       ├───Ground
│   │       ├───GroundParking
│   │       └───Tree
│   ├───Overrides
│   ├───PointClouds
│   ├───Replay
│   ├───Scenario
│   ├───Scenes
│   ├───SeaStates
│   ├───Sensors
│   │   ├───CameraSettings
│   │   ├───FilteredObjectGetter
│   │   ├───GPSSettings
│   │   ├───IMUSettings
│   │   ├───LaserProximitySettings
│   │   ├───LidarSettings
│   │   ├───OccupancyGridSettings
│   │   ├───PlaceableLightSettings
│   │   ├───RadarSettings
│   │   ├───SpeedometerSettings
│   │   └───SplineSensorSettings
│   ├───SituationOverrides
│   ├───Situations
│   └───Weather
```

Figure 12. Example of the folder structure of AILiveSim configuration folders.

2.4 Installation and environment setup

Installation of AILiveSim is simple, the tool is delivered as an archived folder which can simply be extracted to a desired location. For editing the code, the recommended code editor is Visual Studio Code, for which the AILiveSim contains premade workspace- file. The main language used for coding is Python, with the recommended version being Python 3.7, but for this project, version Python 3.8 was used instead without issues.

This was done to have access for the latest versions of Keras-rl2 dependencies, of which all were not working on Python 3.7. The code is still executable on 3.7, but users should note that versions of some supporting libraries might be different. The used version for Keras-rl2 was 1.0.5 and for Tensorflow, the version used was 2.5.0.

The licence file necessary for running AILiveSim is downloaded separately and must be placed into the Licence -folder. For computer vision tasks OpenCV

4.5.2.52 was used. OpenCV can be either installed from source or by using the 3rd -party OpenCV Python wheel. For this project, the 3rd -party package was used to make the installation process faster and easier. As OpenCV played very minor role throughout the project, only being used for one workshop for recording video from camera sensor and adding velocity of the boat as overlaid text to the recorded video, it can be considered an optional library to be installed if needed. PyPlotLib was used for generating the plots for tracking the training progress.

2.5 Workshop scenarios

As a part of the project, four workshops were delivered, intended to showcase features of AILiveSim and to be used as a training material for end-users of the tool. In addition, the workshops also played a role of exercises throughout the project, involving many of the topics and features necessary for the project itself and deepening knowledge of AILiveSim's features and best practices.

The first of the workshops involved an ego boat which was automatically following a set of waypoints represented by buoys, placed in a level representing Turku river. This was implemented by reading the coordinate data from AILiveSim configuration files to get the position of the buoys. Directional vector was then calculated between the boat and the buoy. Buoy positions were saved into a list, from which the coordinates of the next buoy were chosen as the target when to boat got close enough to the currently tracked buoy. This made it possible to have the boat follow a path built using multiple buoys as waypoints.

The coordinates of the buoys were read dynamically from the config file and then added to the list of coordinates, making it possible to dynamically add and remove tracked buoys from the scene without having to edit the code. The second workshop showcased the use of overrides in a simple scenario with two boats on intersecting courses. The scenario contained one ego boat controlled through a Python script. Another boat was following a spline with a speed limit, which was increased between each test run to change the outcome of the scenario. Third workshop demoed the usage of multiple sensors to gather different data.

The workshop had an ego boat equipped with a camera and a speedometer, which were collecting data in a form of a video with speed data overlaid for each frame. OpenCV was used to capture the images and create overlaid text element from the data read from the speedometer sensor. The video was saved as individual images captured 20 times per second and then compiled into a video later using the ffmpeg library. This was done because the video encoder of OpenCV is limited when it comes to supported video file formats, making it difficult to save videos directly through OpenCV.

3. KERAS REINFORCEMENT LEARNING LIBRARY AND REINFORCEMENT LEARNING

Keras-rl is a deep reinforcement learning library, which integrates with the Keras deep learning library to implement different reinforcement learning algorithms. Keras-rl can be extended based on project needs and has inbuilt support for OpenAI Gym, an environment for testing and implementing reinforcement learning algorithms.

Reinforcement learning is a subset of machine learning which is concerned with intelligent agents making decisions in simulated environments. The agents receive rewards if their actions bring the system closer to a solved state, the goal being to maximize the overall accumulated reward during the simulation. This process aims to the discover optimal set or sequence of actions to solve the given problem.

(Sutton S, Barto G, 2018) There are several different algorithms to implement a reinforcement learning network, of which actor-critic model was used in the AILiveSim and Keras-rl integration. More specifically, the actor-critic model is used to teach the boat to avoid obstacles.

3.1 Reinforcement learning and Actor-Critic Model

A reinforcement learning system consists of a policy, a reward signal, and a value function, which in turn define how the agent reacts and adapts to its environment.

The agent in the context of reinforcement learning is the actor with a goal or objective to be solved. Crucial feature of an agent is its ability to sense the environment to adapt and analyse the value of different actions. The agent also has a set of actions which it can execute to get closer to its goal.

The environment is the set of rules or objects in which the agent is operating in. This environment is visible to the agent through different values and variables, like speed or distance to an obstacle, which then give the agent the context with which to make and value different actions it can take. Additionally, a state can also be considered a component of a reinforcement learning system, which represents different values of the agent and environment at a given point in time.

The state-space is the number of possible combinations of different state variables, and the state-space can be either discrete or non-discrete. In a discrete state-space, the variables have quantized values which they can take, with no values between two values being possible. A non-discrete state-space essentially allows for an infinite range of values the variables can take. The number of variables in a state-space denotes the dimensions of the state-space, which is an important consideration for training the model. High-dimensional state-spaces will quickly grow into unmaintainable sizes, especially if non-discrete state-spaces are used.

A policy controls the agent's reactions to different states. The reward signal defines the goal the agent is trying to achieve. The reward signal value tells whether the action helped the agent to move closer to solving the problem or not. While the reward signal signifies the value of actions in short term, the value function measures the value of different actions in the long term. The value function considers not only the current state, but also the states which are likely to follow the current state. This means that certain actions or states which have low value by themselves could still have high value assigned to them by the value function, if those low-value states are likely be followed by a high-value states.

(Sutton S, Barto G, 2018) The actor-critic model is a reinforcement learning algorithm, where policies are independent of the value function. The policy is the actor which selects the actions and value function is the critic, assigning values to the actions proposed by the actor, essentially criticizing it. The critic updates policy distributions based on the perceived values of the proposed actions coming from the actor. (Yoon, 2019) Model is an optional component of reinforcement learning system, modelling the simulation environment. Models are used to plan actions during the learning, predicting the future state and possible actions into the future based on the current state.

Not all machine learning implementations need or use a model, with such systems called model-free systems. These systems are purely based on trial-and-error approach unlike the model-based systems. Some reinforcement learning systems can employ both model-based and model-free methods for learning.

3.2 Learning rate and exploration versus exploitation

Learning rate and exploration versus exploitation are important considerations in a deep learning system. Learning rate is the rate at which the network changes the weights between layer nodes during the episodes or epochs, often represented as a value between 0.0 and 1.0.

A high learning rate means that the network will change the weights more between episodes than it would with lower learning rate. In practice, higher learning rates stabilize on a solution faster, but the resulting solution might not be optimal or stable. A model trained with very high learning rate might have a low success rate or take very long to solve the problem. Training using low learning rate, on the other hand, takes longer to train but the learned solution is generally more accurate than with higher learning rate.

Using too low learning rate might prevent the network from solving the problem at all, essentially causing the model to get stuck forever. The optimal value for learning rate depends on the solved problem, but often used values range between 0.1 and 0.001. Exploration versus exploitation is in many deep learning systems represented by epsilon, again ranging between 1.0 and 0.0. Higher values of epsilon means that the network is more prone to try new ways to approach solution rather than stabilize to a set of previously learned steps to solve the problem, called exploration. A network is said to perform exploitation when it is performing actions based on the previously learned results over trying new approaches at solving the problem.

There exist several optimization methods for exploration versus exploitation, with Epsilon-Greedy being among the simplest of them. In Epsilon-Greedy, most rewarding actions are strongly favoured over less rewarding actions, regardless of the long-term value of those actions. More complex methods can consider the long-term reward value of different actions and states. In general, these methods of optimizing epsilon work by decaying the epsilon as the training progresses so that the network will gradually start preferring exploitation over exploration as more training data is accumulated over the episodes. (Salloum, Z. 2019.)

3.3 Collision Avoidance

Collision avoidance systems are developed to prevent collisions between vessels and vehicles by employing data from sensors to either warn the controllers of the impending collision or even take control of the vehicle and automatically perform the necessary control actions to prevent the collision.

Some collision avoidance systems rely on simply measuring the surroundings to try and detect threatening objects, while some systems also rely on systems on different vehicles communicating with each other. (Seiler, Song & Hedrick, 1998) For example, the TCAS (Traffic alert and collision avoidance system) used in aircraft to avoid mid-air collisions relies on the TCAS's on the approaching aircraft communicating between each other to resolve the correct action to take. After resolving the action, TCAS's in the planes give orders to the pilots to take different actions to avoid the collision, usually by ordering one of the planes to descend and the other to descend.

(Livadas, Lygeros & Lynch, 2000) Similar solutions are also used in maritime sector to prevent collisions between ships and other maritime vessels. These systems usually employ sensors like radar and lidar to detect objects which are in collision course with the ship. Data gathered by these sensors can also be used as an input for a reinforcement learning system, where reinforcement learning can be used to teach the system to detect different hazardous situations and react accordingly. The data fed in can be simple, like distance from other objects or more complex, like headings and velocities of nearby vessels which can then be used to calculate the closest distance between the ships will be from each other in the future. For this project, a simple approach using only distance from obstacle was chosen to make the training phase shorter.

3.4 Weights and layers

Weights are a set of values which the neural network learns during training phases. These weights describe the strength of connection between neurons of the neural networks, sometimes also called nodes. The weight value decides how much a given input affects the output.

A similar concept to weight is bias, which is also a learnable parameter within a neural network. In contrast to weight, bias describes the distance between the predicted values and intended values in the neural network. A weight file is a file used to store the values of weights and biases in a given network, which can be then used to transfer the learned values forward for further training with different network or to store the weights for later use, so that the network can be resumed from the checkpoint using the weights at a later time. Layers are collections of nodes or neurons, through which the data travels and is manipulated as the network learns by altering the weight values to reach an optimal solution.

A singular node within neural network represents a mathematical function which alter the input data or a value of a variable. The layers can be broadly divided in three categories, the first of which is Input layer. Input layer contains the raw data which is used to perform the learning. The neurons in this layer represents an attribute from the training dataset, used to train the model. Second layer is called the hidden layer, of which there are usually several. Hidden layers are the place where the learning happens, where the neurons of these layers take in the data from input layer and use activation function to alter it to either get closer to solution or further from it, in which case the network will readjust itself.

The number of hidden layers varies between implementation, where more complex solution tends to have higher number of hidden layers than simple implementations. A lower number of hidden layers makes the training process faster, as there are less combinations on input to learn but also reduces the expressiveness of the network. This means that a network with low number of hidden layers might not be able to find as good a solution when compared to a network with higher hidden layer count, only limiting itself to somewhat good set of actions. A network with higher layer count, conversely, will take longer to train, but is generally able to find a more optimal solution compared to the low-layer count network.

Hidden layers can be either fully connected, where all nodes from previous layer are connected to nodes of current layer, or convoluted, where nodes only connect to a subset of nodes nearby, forming a local group of nodes. In general, convoluted networks are more efficient than fully connected ones. In practice, many convoluted network implementations also feature fully connected layers. There is no one absolute theory or formula to calculate optimal value for the number of hidden layers and nodes per layer, so the structure and depth of the hidden layers must be based on testing and experiences drawn from other similar implementations with varying levels of complexity used as a measuring scale for the number of layers needed.

4. INTEGRATION OF KERAS REINFORCEMENT LEARNING INTO AILIVESIM

This chapter outlines the process of integrating Keras reinforcement learning library into the AILiveSim environment. The test scenario and the setup will also be explained here, as will the justification for choosing Keras instead of other RL libraries.

One reason for choosing Keras was the fact that there was an existing experience base on using Keras for implementing reinforcement learning on a very similar environment. Moreover, Keras supports all the features needed for implementing the desired functionality for the test case, while also having good documentation available if needed.

One test epoch consists of 400 timesteps with a length of 0.1 seconds. Based on test runs lasting for 4-5 hours, a value of 0.01 was chosen for learning rate, while epsilon decay was set to 0.002. These values led to a stable result with reasonable training time. The time used to train the model with se aforementioned settings was around 30 hours, but the model stabilized to a decent solution long before that, achieving a hit rate of around 60% after 6 hours of training. Hit rate was calculated by simply dividing the number of successful runs with total number of episodes. A run was considered successful if the boat reached a distance of 3000, corresponding to around 30 real-life meters, units to the goal.

4.1 Test case and implementation

The chosen test scenario for the reinforcement learning implementation was a simple collision avoidance situation, where a moving boat tries to avoid a collision with a stationary larger boat. The scene also contains a waypoint, towards which the moving boat tries to steer.

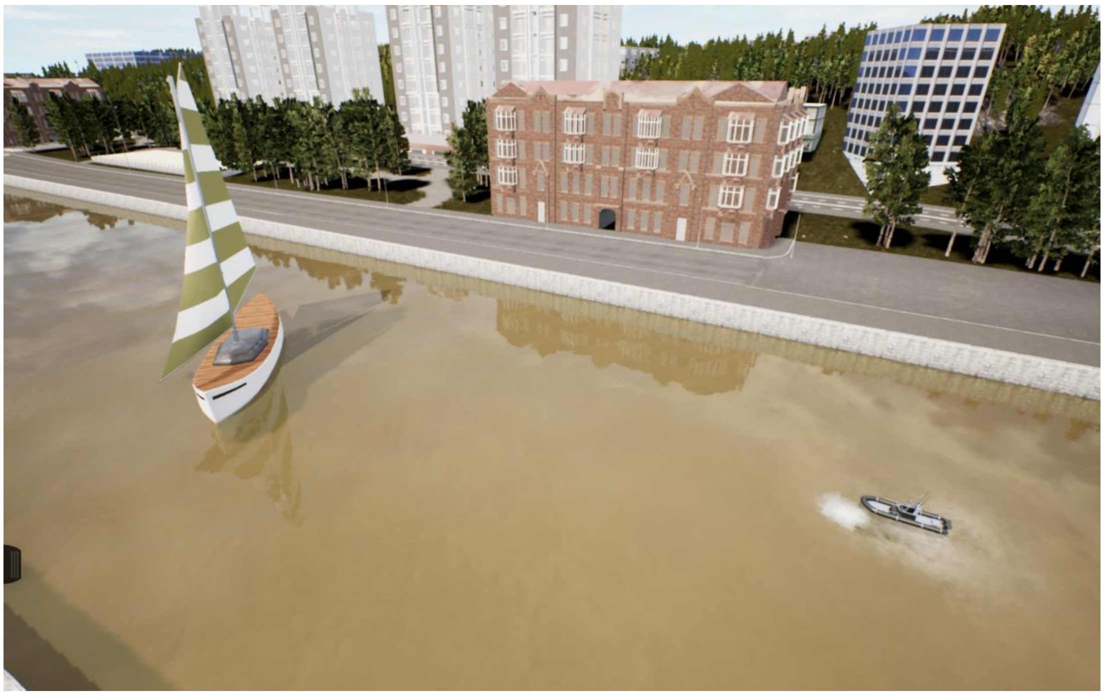


Figure 13. The test scenario. The moving boat to the right and the obstacle in the left. The waypoint is behind the larger boat.

The chosen method to achieve the reinforcement learning was the Actor-Critic model, where the moving boat is rewarded when it gets closer to the waypoint and penalized whenever it collides with either the obstacle or the riverbank.

The AI Gym environment was chosen as the learning environment framework. The reason for this choice was that AI Gym has good integration with Keras and is also commonly used in many reinforcement learning applications. For training purposes, the same situation was also recreated at a more spacious part of turku river to give room for the boat to avoid the obstacle without colliding into riverbanks. This was problematic, because the learning episodes were terminated before the boat was able to traverse beyond the obstacle, making it difficult to learn the correct weights for the actions to avoid the obstacle.

The second, more complex training environment consisted of a cuboid representing the föri ferry, moving from riverbank to riverbank (Figure 14). The ferry is placed between the boat and the target, just like in the previous scenario, with an animation spline used to control the ferry. The goal was the same as before, to reach the red target without colliding with either the ferry or riverbanks.

When setting up the scenario, the scale of the controlled boat and river was mismatched, with the boat being slightly too large in comparison to the river. To amend this, the boat was scaled down by around factor of two, essentially halving its size in all dimensions.

This was done to make the situation more representative of a real scenario and to ensure that the boat would have enough space to manoeuvre around the riverbends and the obstacle. To accommodate this change, the throttle value was halved to 0.5 as were the steering command, with their values set to 0.5. This was done to make the downscaled boat behave smoother by making the movements less erratic and fast.

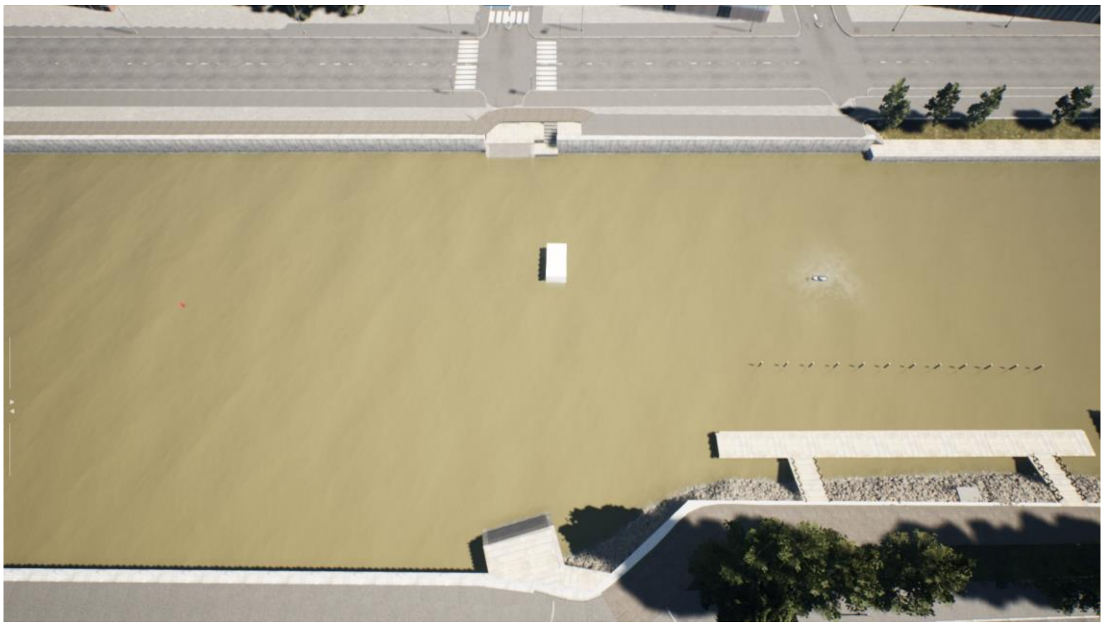
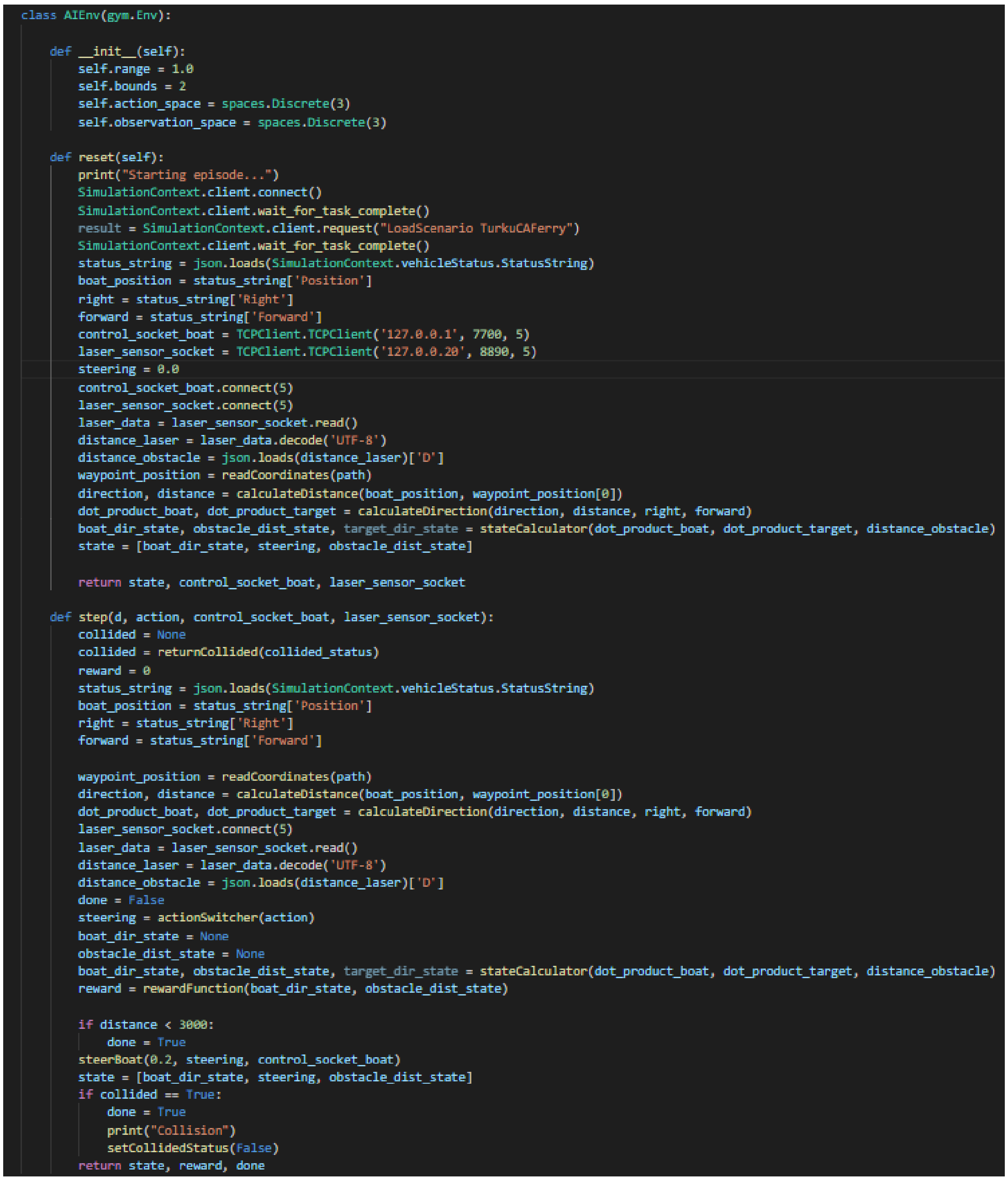


Figure 14. The more advanced scenario with moving obstacle in the middle, represented by a white cuboid. The controlled boat is on the left and the goals is to the right, represented by the red buoy.

4.2 Integration process

AILiveSim does not have inbuilt AI Gym integration, so to interface AILiveSim and Keras together using AI Gym, a custom AI Gym environment is required. This is achieved by declaring a Python class with the gym.Env parameter (Figure 15).



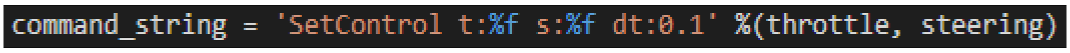
```python
class AIEnv(gym.Env):

    def __init__(self):
        self.range = 1.0
        self.bounds = 2
        self.action_space = spaces.Discrete(3)
        self.observation_space = spaces.Discrete(3)

    def reset(self):
        print("Starting episode...")
        SimulationContext.client.connect()
        SimulationContext.client.wait_for_task_complete()
        result = SimulationContext.client.request("LoadScenario TurkuCAFerry")
        SimulationContext.client.wait_for_task_complete()
        status_string = json.loads(SimulationContext.vehicleStatus.StatusString)
        boat_position = status_string['Position']
        right = status_string['Right']
        forward = status_string['Forward']
        control_socket_boat = TCPClient.TCPClient('127.0.0.1', 7700, 5)
        laser_sensor_socket = TCPClient.TCPClient('127.0.0.20', 8890, 5)
        steering = 0.0
        control_socket_boat.connect(5)
        laser_sensor_socket.connect(5)
        laser_data = laser_sensor_socket.read()
        distance_laser = laser_data.decode('UTF-8')
        distance_obstacle = json.loads(distance_laser)['D']
        waypoint_position = readCoordinates(path)
        direction, distance = calculateDistance(boat_position, waypoint_position[0])
        dot_product_boat, dot_product_target = calculateDirection(direction, distance, right, forward)
        boat_dir_state, obstacle_dist_state, target_dir_state = stateCalculator(dot_product_boat, dot_product_target, distance_obstacle)
        state = [boat_dir_state, steering, obstacle_dist_state]

        return state, control_socket_boat, laser_sensor_socket

    def step(d, action, control_socket_boat, laser_sensor_socket):
        collided = None
        collided = returnCollided(collided_status)
        reward = 0
        status_string = json.loads(SimulationContext.vehicleStatus.StatusString)
        boat_position = status_string['Position']
        right = status_string['Right']
        forward = status_string['Forward']

        waypoint_position = readCoordinates(path)
        direction, distance = calculateDistance(boat_position, waypoint_position[0])
        dot_product_boat, dot_product_target = calculateDirection(direction, distance, right, forward)
        laser_sensor_socket.connect(5)
        laser_data = laser_sensor_socket.read()
        distance_laser = laser_data.decode('UTF-8')
        distance_obstacle = json.loads(distance_laser)['D']
        done = False
        steering = actionSwitcher(action)
        boat_dir_state = None
        obstacle_dist_state = None
        boat_dir_state, obstacle_dist_state, target_dir_state = stateCalculator(dot_product_boat, dot_product_target, distance_obstacle)
        reward = rewardFunction(boat_dir_state, obstacle_dist_state)

        if distance < 3000:
            done = True
        steerBoat(0.2, steering, control_socket_boat)
        state = [boat_dir_state, steering, obstacle_dist_state]
        if collided == True:
            done = True
            print("Collision")
            setCollidedStatus(False)
        return state, reward, done
```

Figure 15. Example of the AI Gym custom environment with the custom step and reset functions.

An AI Gym environment requires two functions to operate: reset and step. The AILiveSim functionality necessary for running the simulation and controlling the objects is implemented in these functions so that they can be operated from the Keras network.

The reset- function is used to restart the simulation after an episode has finished. This function should contain the logic for loading a scenario and establishing variables needed to track objects in the scenario as well as establish a starting state for an episode. Step is used to progress the simulation within an episode and to apply commands to the boat.

For this purpose, AILiveSim allows usage of the deltatime operator on the control commands to allow executing a given control command for a limited time. OpenAI Gym implementations also use render -function to return data from the simulation environment. This data can be used to monitor the progress of the learning process and troubleshoot possible issues. This function is not necessary for integration with AILiveSim, as the tool already has a 3D interface which gives good representation of the ongoing training process and the agent's behaviour.



```
command_string = 'SetControl t:%f s:%f dt:0.1' %(throttle, steering)
```

Figure 16. Example of using deltatime, dt, to limit the execution time of a command to 0.1 seconds.

To steer the ship, a command string with a deltatime operator is used. This ensures that on each step, the given control command is only executed for the duration of the step. The reward function grants a reward based on the direction compared to the target, while avoidance still uses distance-based implementation.

The heading in relation to the target is calculated using dot products calculated from boat to target, and target to boat. These dot products are calculated using the position of the boat and the target and a variable returned by AILiveSim status string called Right. Using only the dot product calculated between boat and target is not sufficient, since the value of the product ends up with value of the zero both when facing the target directly or when facing directly away from it.

This can be remedied by using the dot product calculated between the target and boat, because this dot product will conversely take values between 1 and -1 when facing the target or directly way from it, respectively. If the value of the target-boat dot product is negative, the boat is facing away from the target. There are nine rewarding states for the heading of the controlled boat based on these dot products, four for the left and right headings, and another for facing away from the target. The first state is the most desired one, having a reward value of 100. This state triggered if the boat is facing almost directly towards the waypoint.

There are seven other states with lower rewards which are triggered if the boat is facing the target but is not directly facing it, with a wider angle corresponding with a lower reward. The final state is triggered if the boat is facing away from the target, which has reward value of 0. These rewards are calculated per timestep, which in the testing implementation had a length of 0.1 seconds. The collision state is handled with two states, 0 and 1. 0 is the state where the boat is far away from the obstacle and there is no risk of collision. Every timestep the boat is in state 0 the reward is unchanged. State 1 is triggered if the boat is withing the range of 4000 units from the obstacle, and in this state each timestep spent deducts 180 from the reward.

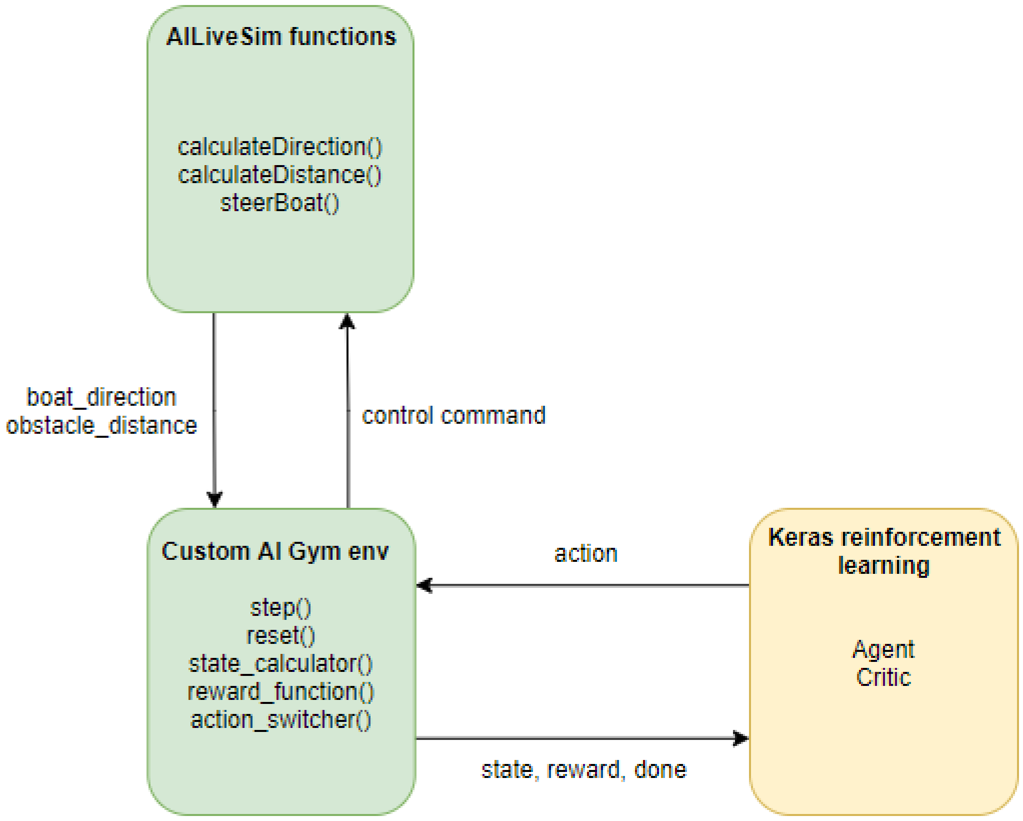


Figure 17. A rough overview of the relationship between components.

The custom AI Gym environment is a subclass of TurkuRL, which contains all the functions used in the AI Gym environment to calculate distance and direction of the boat. Functions used by the AI Gym environment reside outside of the AI Gym class, from where they can be called as necessary. This approach ensures that variables and AILiveSim classes like VehicleStatus and SimulationStatus are available for the functions. Actions are chosen randomly by the Keras network, which are represented by integers between 0 and 2. These integers are then converted into proper control commands by the action_switcher function, which maps each integer to a steering value.

Boat's position and heading in relation to the target are calculated once per timestep as is the distance to the obstacle. These variables are then passed to the state_calculator function to generate discretized values for reward_function. Inside reward_function the discretized states are assigned reward values. These reward values are then passed to Keras-rl implementation alongside with the discretized states. In addition to state and reward -variables, there is also a special done -variable, which is used to notify Keras-rl that the current episode has ended. done is a Boolean variable which starts as false in the beginning of each episode. The value of done is changed to true on predefined events, which are intended as endpoints of the training episodes.

In the case of AILiveSim collision avoidance implementation, such endpoints were a collision with any object in the scene or reaching a distance of 6000 units from the destination. After the value of done is set to true, Keras-rl will end the running episode and call the reset -function to start a new one. The state, reward and done are sent once per timestep, after which Keras-rl takes care of assigning values to different actions and trying to predict the best actions to take based on the history of action. The random distribution of actions is then weighed towards the optimal ones, and these are then sent to the AI Gym env to be translated to control commands.

The state -variable is an array, which contains three variables. These variables are discretized heading, steering, and an integer flag denoting whether the boat is dangerously close to an obstacle. In addition, whenever the boat reaches a distance of 3000 units from the goal, a variable named success will be set to true. This variable will be checked by the reinforcement learning code whenever the status of done is true, and if the value of success is true, number of hits will be increased by one. This variable is used to calculate the number of times the boat managed to reach the goal versus the number of failed attempts at reaching it. This value is not used for training the model, its only purpose being debugging and tracking of the training process. For further progress tracking, a plot was printed every 10 episodes with the cumulative rewards over the training episodes. Having this data available made analysing the progress of the training easier. If the cumulative score or hit rate was not increasing after a prolonged training period, this could hint that the model implementation was inadequate, whereas constant increases over time meant that the model was learning and improving.

4.3 Reinforcement learning model implementation

The reinforcement learning model was implemented as an actor-critic model using keras-rl. Number of inputs and actions are represented as dense keras layers, where a softmax policy is used for choosing actions to take. Lists are used to store histories for rewards, critic values, and action probabilities for learning. The reset -function sets the initial state and establishes the socket connection for the boat and the sensor.

Therefore, reset returns control sockets for both the sensor and the boat, which are then passed as parameters to the step -function, which is responsible for advancing the learning and simulation process. By establishing the connection sockets during the beginning of each episode and then reusing the sockets for each step helps prevent disconnections and improves performance by not having the system establish multiple TCP connections per episode. In addition to the sockets, reset will also return the initial state to be used as a starting point for the first step.

The step function uses the initial state to calculate the reward and a new state to be used for the next step. Step will also return the Boolean variable done, which tells the network whether the training episode should proceed. If the variable is false, the episode will be ended, and both the sensor and boat sockets will be disconnected to prevent having multiple overlapping TCP connections.

After the training episode reaches the end, the rewards will be normalized, and these normalized values will be used to calculate the actor and critic losses and saved into an array representing the memory of the model. The episodic reward and action histories will then be cleared in preparation for the next episode.

4.4 Testing

The implementation and integration were tested in a test scenario made specifically for this purpose. The scenario consists of a waypoint, a controlled boat, and an obstacle. The obstacle is placed between the controlled boat and the waypoint, with the controlled boat facing directly at the waypoint and the obstacle.

The goal of the scenario was for the boat to reach the waypoint while avoiding collision with the obstacle. The tested algorithm uses discrete state-space consisting of heading to the target, steering value and collision state, creating a three-dimensional state-space. A discrete state space based on increments in steering proved to be difficult to implement due to how steering commands are implemented in AILiveSim.

For example, to increment and decrement steering by 0.2, a mechanism to check whether the steering is already at maximum value of -1.0 or 1.0 is necessary. It is also recommended to make sure that the agent can recognize if maximum steering has been achieved, so that the same action will not be unnecessarily repeated. These complications made it more difficult to verify the functionality of the model and made it harder to troubleshoot any possible issues arising due to incorrect mapping of actions.

To combat this, steering was not handled with increments but with absolute values instead. These discretised headings were then used to send the state back to the network. These states correlate with the reward function, with states derived from certain combination of reward and heading correlating with the state sent back to the network. The action space consisted of three actions, full turn to the left, full turn to right and going straight.

Original plan was to have five actions, with half-turns to left and right included as well, but those actions were found redundant as alternating the full turns already fulfilled the function of fine-tuning steering. Removing the half-turns reduced the state space and sped up the learning phase. Next phase was to use the system in a more complex setting involving moving obstacle in Aura river.

The scenario simulates a boat moving through Aura river while trying to avoid the ferry moving between the riverbanks. The model trained on the scenario with static obstacle was not directly applicable for the more complex scenario due to assumptions made in the training phase. These stemmed from incorrect training parameters, one of which was the distance between the obstacle and boat.

The training scenario assumed a range of 4000 being dangerously close, but in the actual situation a range of 4000 proved to be too long, as the river itself did not have enough room to avoid the obstacle from that far away. Another issue was avoidance of riverbank collisions. In the training scenario, the boat was far enough from the riverbank so that collisions were extremely rare.

When moved into a much more confined space, the model was not always able to correctly detect dangerous states with riverbanks. To remedy this, the range calculation was changed from position-based system to a system using a proximity laser sensor. The sensor is among the many sensors directly implemented in AILiveSim, and it works by scanning obstacles within a given range. If the sensor does not detect an obstacle within the range, the return value is 0.0 and if an object is detected the range to that object will be returned.

The model used in collision avoidance was changed so that if the returned value from the sensor was 0.0, the flag denoting collision state was set to 0, meaning that the boat was within safe distance. If a distance value other than 0.0 was returned the boat was considered to be within a dangerous distance from an obstacle, and the flag was set to 1. The range from which the sensor detects obstacles was set to 2000 for the purposes of training the model. This also meant that the model had to be retrained on the new scenario, as the modelled situation had gone through extensive changes compared to the simpler training scenario. Biggest change was the addition of riverbanks which had to be avoided, which the old model was unable to detect and consider when choosing actions.

4.5 Challenges

One challenge faced during the integration was the time consumed by the training process. Headless training without the 3D interface of AILiveSim was not possible with the standalone version used in the thesis project, meaning that the resource consuming 3D graphics had to be run during the training. This made it impossible to execute the learning phase at highly accelerated speeds due to physics breaking down if the simulation was running at speed factors exceeding 6. This means that all training had to be done with nearly real-time timescale. Another issue was the handling of the TCP connection during extended testing periods, where an unresponsive socket could either crash the training process outright or cause the state of the boat not to update, corrupting the generated weights and making them unusable.

The issue of resource intense 3D interface was solved by setting up another computer solely for the training process, which could be kept running overnight. This allowed dedicating resources of the training computer for the training alone, freeing up the main system for development purposes. The TCP errors were fixed by altering the way the connections are handled during training. Previously, connection was established on each step, which worked but caused issues with reliability and speed. After changing the design so that the connection is established only once at the start of an epoch, the connection errors were remedied. One important factor for successful testing is the availability of internet connection.

AILiveSim requires internet connection to verify the licence, meaning that losing connection will eventually crash the software. Thus, to successfully perform the training process, the system must have a stable internet connection.

4.6 Training

Based on the experience accumulated during the testing and training phase, some guidelines can be drawn to perform the training efficiently. First off, the system used for training should be decently powerful so that the 3D Interface of AILiveSim can be ran with high enough framerate.

A decent gaming PC should meet the requirements without much issue, the system used for training the model used in the project used a 5-year-old quad-core CPU and a GTX 1080 GPU for good results.

Memory is not a major factor, with 8 gigabytes being enough for a training system. A stable internet connection is crucial for the testing setup as AILiveSim requires access to internet to verify the licence.

If internet access is lost at any time, AILiveSim will halt and stop executing the training scenario. As the training time for more complicated collision avoidance scenario was around 30 hours, this factor is extremely important to prevent loss of training progress.

4.7 Keras RL Details

There exists multiple libraries and frameworks to implement the described machine learning functionality using Python as the programming language. For this project, KerasRL was chosen mainly due to pre-existing knowledge and experience base within the working group.

After preliminary examination of different frameworks and libraries, KerasRL was found to have all the necessary features to achieve the needed reinforcement learning functionality. The main choice was between KerasRL, Pytorch and Tensorforce. All the three libraries had comparable feature sets and adequate set of supported algorithms, and the documentation between the three was also considered very good. The main differentiator therefore was familiarity with each tool, where KerasRL had and edge. KerasRL had lots of third-party documentation available on different projects and implementations, made by people and institutions. Moreover, people related to the project also had existing experience to drawn upon using Keras and KerasRL.

This led to the decision to use KerasRL over the other options. There also exist other frameworks and libraries in addition to the mentioned three options, but these were found to have lacking documentation at the time of writing, so they were not considered further. An important note about KerasRL is to use the version 2 of the framework, as the development for the first version was ceased at the time of writing.

The second version of KerasRL is in active development and supports Tensorflow 2.0 and beyond, ensuring good compatibility. The implementation presented in this document can be achieved reasonably well with any machine learning library, but this would require adapting the integration method to suit the chosen library, while the theory of algorithms and other fundamental machine learning factors should stay the same. The weights of the network are saved as md5 -files, which can be loaded using Keras' own load_weights -function. These weight files were by default saved every ten episodes to create checkpoints from which to resume the training. These files were loaded at the beginning of training sessions and then saved by overwriting the old files automatically, so that the weights build up as training time passes.

This kind of approach was very useful due to the long training times necessary to train the model. This approach allowed the training to be stopped in case of thunderstorms or other events posing risk to the hardware without ever losing more than 10 episodes worth of progress in the worst case. The weight files consist of three different files, a data file, index file and finally, the checkpoint file. When saving the weights, the save_weights function should be used by giving it a folder as a parameter and not a full filename, so that Keras will correctly create all the three files. When loading the weights, the path should point to the filename of the weights without file extension to make sure Keras-rl correctly loads the data from all the three files. The files can be stored and transferred anywhere, but the project code assumes they reside within the AILiveSim project folder.

Some issues were faced when trying to transfer the weights between different systems, where the loaded weights would incorrectly lead the model to spin the boat endlessly rather than correctly avoid the obstacle and reach the checkpoint. This was not reproducible using the two systems used for training and coding, but the issue manifested when ran on the production system. No apparent cause was found for this, but it could be related to some minute differences between how the neural network is established when using certain pieces of hardware or some background processes preventing AILiveSim from properly setting up the control sockets. The training itself ran fine on the production system, so the issue was circumvented by retraining using it.

5. RESULTS AND DISCUSSION

The starting point of the project was to integrate reinforcement learning functionality into a simulation tool AILiveSim, with the goal of providing a proof-of-concept implementation for possible future reinforcement learning approaches using AILiveSim.

The way the integration between Keras and AILiveSim was implemented was by using a custom AI Gym class implementing custom versions of Keras functions step and reset. The result of the project was a reinforcement learning system based on Keras-rl successfully running within AILiveSim.

The implementation using the custom AI Gym class proved to be a good and straightforward way to integrate reinforcement learning into AILiveSim, allowing the execution of reinforcement learning models in AILiveSim. The system was able to learn the collision avoidance algorithm to an acceptable degree, avoiding the obstacle for most of the time while also being adaptable enough to be applied on similar situation on different environment. When moved to a scenario with moving obstacle and more confined space, some retraining and remodelling was necessary due to the new sensor added to the boat.

With more time to train, the results could have been even better, but due to the requirement to run the training in real time, the workflow to training and improving based on the previous results ended up consuming most of the time during the training phase. The integration can also be used to implement other reinforcement learning applications for use-cases other than collision avoidance by e.g., using data from sensors optimize for time or route length using reinforcement learning.

The system was adapted by adding a new sensor to detect riverbanks, which was initially a problem when used in a different scenario. The integration based on custom AI Gym environment was adaptable enough to implement the new distance-based state system with the sensor with relatively minor changes to the environment. The used actor-critic model was directly based on Keras example documentation, ensuring that the model implementation did not bottleneck the integration functionality. Likewise, necessary considerations set forth by the background theory were also fulfilled with the chosen method.

Using the custom AIGym class to implement the integration does not limit the ability to use different algorithms and machine learning approaches significantly. As long as the design based around step and reset -functions is adhered to, the machine learning implementation can be implemented as wished. One limitation of the approach is the need to modify the two functions if the machine learning code is modified. This is because the two functions need to have necessary functions and variables to deal with the data fed from the neural network and if this data is changed, the functions must be changed accordingly. When it comes to the adherence of fundamental machine learning theory, the approach used in this project does not pose any limitations in that regard. As mentioned previously, the used approach for machine learning can be freely customised as long as the step and reset functions are modified in accordance. The system also supports saving and reusing weights derived from the training sessions.

AILiveSim had multiple updates adding features and fixing bugs throughout the project, with latest version used for development being 1.5.27. The machine learning integration should not be affected by the used version of AILiveSim. The updates were done as clean updates whenever possible, by removing the old version and then installing the latest one. Code was then pulled from a GitHub repository into the new installation. This was done to prevent any conflicts arising from overwriting old files with new ones. Most of the features used in building the scenarios were left intact between version updates, with some exceptions. Notably, the speed limit of spline objects was not functional in versions prior to 1.25, so animation spline was used instead of a normal spline for moving the ferry. With later versions the animation could be replaced with a normal spline if needed.

In general, AILiveSim, at the time of writing, is constantly being updated with new features and bugfixes, so in the future some of the methods and features detailed in this document might be outdated and some features might have more efficient ways to configure and set them up. All of the descriptions features and methods used to achieve certain functionality are based on AILiveSim versions

1.5.27 and before, with the focus on being firmly on the reinforcement learning integration.

Readers seeking to base their work on this document should review the feature-set of their version and adapt the approach to work with those features and updates where necessary.

5.1 Usage of frameworks other than Keras-rl2

While the implementation in this thesis is technically achievable with any machine learning framework for Python, the steps and method specified here are specific to Keras-rl2 and are not directly applicable when using other frameworks.

The implementation in this project relies on the step and reset functions, implementation of which is specific to Keras-rl. If use of other framework is desirable, the integration between the framework and AILiveSim must be redone according to the chosen framework's specification, essentially replacing the solution using step and reset functions specified in this thesis with functions or API suitable for use with the chosen framework. Other core concepts should be similar between the frameworks, there might be some differences between available optimizers and algorithms which could affect the optimal values of variables like epsilon and learning rate, but the core principles of setting the values for the variables and modifying learning rate will remain the same.

AILiveSim-specific features, like reading data from sensor data and translating control commands between the neural network and AILiveSim should generally work the same, no matter the used framework. Some alterations for the translation function might be needed, depending on how the chosen network handles actions, but the principle of tying a certain action value to a steering value in AILiveSim stays the same.

The thesis also assumes integration implemented using OpenAI Gym, so if a different integration method is desired, the steps and methods outlined in this document will not suffice as-is. Integration methods beyond using AIGym were not within the scope of this project, so if library other than AIGym is used, the research and technical implementation of such integration is left to the reader.

5.2 Notes on training

As mentioned previously, the training of the model had to be done on real time due to the non-could version of AILiveSim not supporting headless training.

A setup that was found out to be functional for this was to use two systems, of which the more capable one is used for training the model and any other system for doing the code and editing the AILiveSim scenarios. This way, the training system could be placed to a location where it could be conveniently run for multiple days at a time, without the sound or heat from the system causing inconvenience, while also making it easy to serve a stable wired connection to the internet without the need to resort to wireless connections. Using different system for editing also made the editing more fluid, as the high resource drain of the training process would bog down the system if anything else was done on it.

This setup was relevant especially due to the ongoing Covid19 -pandemic, due to which office environment was not available at any point during this project, which made it especially important to have at least some kind of PC available for other daily tasks while the training system was dedicated to training the model. Even in normal office environment, many of the above-mentioned points on using dedicated training system still apply.

One benefit of the from-home setup was the ability to access the training system at any time without needing to configure any kind of remote-access setups, which would be needed for office-based system if access is needed outside of office hours. Also, all of the points mentioned here apply only for the real-time training. If at any point AILiveSim will support training headlessly, without the 3D UI, that will render at least some of the points mentioned here irrelevant, but without access to such feature at the time of writing, those possible differences are not elaborated further here.

5.3 Future development

Future development opportunity is to integrate data from sensors and use that data to generate behaviours using the gathered data. More opportunities to leverage reinforcement learning could be found, for example, by using GPS data for route and path optimization.

The implemented collision avoidance algorithm can be improved by using lidar and radar data to implement more robust algorithm for avoiding collisions by using not only the position of the boat and the obstacle, but also the speed and headings of the surrounding vessels.

REFERENCES

The Best Tools for Reinforcement Learning in Python You Actually Want to Try (2020). Accessed on 11.08.2021

Blueprints Visual Scripting. *https://docs.unrealengine.com/en-US/ProgrammingAndScripting/Blueprints/index.html* (Dec 4, 2020). Accessed 17.04.2021

Github - keras-rl/keras-rl: Deep Reinforcement Learning for Keras. Available: https://github.com/keras-rl/keras-rl. Accessed 17.04.2021

Keras: The Python Deep Learning APIc. Available: keras.io. Accessed 18.04.2021

Programming Basics. *https://docs.unrealengine.com/en-US/ProgrammingAndScripting/ProgrammingWithCPP/Basics/index.html* (Dec 4, 2020). Accessed 17.04.2021

Programming with C++. *https://docs.unrealengine.com/en-US/ProgrammingAndScripting/ProgrammingWithCPP/index.html* (Dec 4, 2020). Accessed 17.04.2021

PyTorch [2021, Aug 11]

Tensorforce: a Tensorflow library

Weights and Biases - AI Wiki [2019, 17.08]

Alammar, J. A Visual and Interactive Guide to the Basics of Neural Networks [2021, Aug 17,].

Alpaydin, E. & Bach, F. (2014), *Introduction to Machine Learning,* MIT Press, Cambridge, UNITED STATES.

Brownlee, J. 2019, Understand the Impact of Learning Rate on Neural Network Performance.

Leudet, Jérôme & Christophe, Fraçois & Mikkonen, Tommi & Männistö, Tomi. (2019) *AILiveSim: An Extensible Virtual Enviroment for Training Autonomous Vehicles.*

Livadas, C., Lygeros, J. & Lynch, N.A. 2000, "High-level modeling and analysis of the traffic alert and collision avoidance system (TCAS)", Proceedings of the IEEE, vol. 88, no. 7, pp. 926-948.

Ross, Sheldon M.. *Simulation*, Elsevier Science & Technology, (2012). *ProQuest Ebook Central*, https://ebookcentral-proquest-com.ezproxy.vasa.abo.fi/lib/abo-ebooks/detail.action?docID=1044919.

SALLOUM, Z. 2019, -04-24T23:26:22.585Z-last update, Exploration in Reinforcement Learning. Available: https://towardsdatascience.com/exploration-in-reinforcement-learning-e59ec7eeaa75 [2021, Jun 9].

Seiler, P., Song, B. & Hedrick, J.K. 1998, "Development of a Collision Avoidance System", SAE Transactions, vol. 107, pp. 1334-1340.

Suqiyama, M. 2015, Statistical Reinforcement Learning : Modern Machine Learning Approaches, Chapman and Hall/CRC.

Sutton S, Richard & Barto G, Andrew. (2018), *Reinforcement learning, an introduction.*

Sandford, P.G. (2010), Simulation in Nursing Education: A Review of the Research - ProQuest. https://search.proquest.com/openview/84e666ec7b4dbf8ff8d434ac009fc267/1?pq-origsite=gscholar&cbl=55152

Oliver, Dean S & Chen, Yan (2020), *Recent progress in machine learning applications in reservoir simulation*, Gulf Professional Publishing.

Yoon, Chris (2019), Towards Data Science, Understanding Actor Critic Methods and A2C | by Chris Yoon | Towards Data Science, Accessed 18.04.2021