## TASK DESCRIPTION

**Budapest University of Technology and Economics**
Faculty of Electrical Engineering and Informatics
Department of Control Engineering and Information Technology

Megalla Antony Tharwat Fekry

# CAR SIMULATOR TEST ENVIRONMENT FOR ARTIFICIAL NEURON SYSTEM DEVELOPMENT

TDK Challenge Work

SUPERVISOR

Dr PETER NEUMANN

BUDAPEST, 2023

## Contents

## Abstract

The Department of Electron Devices (Semiconductor Technology Laboratory) is conducting ongoing research projects on implementing Physical Artificial Neurons (ANs) using Vanadium Oxide (VO2). [1]– [3] The networked ANs allow us to get closer to the brain-like operation devices. Understanding the HW-based ANs operation requires a stable test environment. [1]

This research focuses on designing a flexible Artificial Neural Networks (ANN) test environment. One interesting engineering problem is a self-driven car study. A key point of HW-based ANs' development is using a self-driven car simulator to examine different types of ANNs' behavior and operation in a simulated, well-defined environment. [2]

To address the need for energy-efficient ANs, this work employs an STM32 microcontroller unit (MCU) based on the Arm Cortex-M processor. A Python-based car simulator is developed using the PyGame library, featuring manual and self-driving modes. The Neural Network is created for the self-driving car simulator by STM32, and two-way communication protocols are analyzed between the MCU-based ANN and PC simulator.

The self-driving car simulator successfully measures distances using radar sensors and translates steering commands into vehicle directions. The communication between the deployed neural network model on the STM32 MCU and the PC car simulator is established, facilitating data exchange. [3]

This research addresses the challenge of testing evolving hardware in real-time environments and emphasizes simplifying algorithmic systems and data gathering. The study also highlights the importance of converting data between digital and analogue formats for seamless communication. Successful completion of this work contributes to the testing of physical ANN hardware and efficient communication protocols for real-time deployment.

Keywords:

Physical Artificial Neuron, Self-Driven Car Simulator, Communication Protocols, STM32, MCU, Neural Network, Real-Time Environments.

## الملخص (Arabic Abstract)

تقوم إدارة الأجهزة الإلكترونية (مختبر تقنية أشباه الموصلات) بإجراء مشاريع بحثية مستمرة حول تنفيذ الخلايا العصبية الاصطناعية الفيزيائية (ANs) باستخدام أكسيد الفاناديوم .(VO2) تسمح لنا الشبكات العصبية الاصطناعية الشبكية بالاقتراب أكثر من أجهزة التشغيل الشبيهة بالدماغ. يتطلب فهم تشغيل الخلايا العصبية الاصطناعية القائمة على الأجهزة الصلبة بيئة اختبار مستقرة.

تركز هذه الدراسة على تصميم بيئة اختبار مرنة للشبكات العصبية الاصطناعية (ANNs) . تتمثل إحدى التحديات الهندسة المثيرة للاهتمام في دراسة السيارة ذاتية القيادة. تتمثل إحدى النقاط الأساسية في تطوير الشبكات العصبية الاصطناعية القائمة على الأجهزة الصلبة في استخدام محاكاة السيارة ذاتية القيادة لفحص أنواع مختلفة من سلوك وتشغيل الشبكات العصبية الاصطناعية في بيئة محاكاة محددة جيدًا.

و نظرًا للحاجة إلى خلايا عصبية اصطناعية موفرة للطاقة، يستخدم هذا العمل وحدة تحكم دقيقة (MCU) STM32 تستند إلى معالج .Arm Cortex-M تم تطوير محاكاة سيارة تعتمد على Python باستخدام مكتبة PyGame، والتي تتميز بالوضعين اليدوي وذاتي القيادة. تم إنشاء الشبكة العصبية لمحاكاة السيارة ذاتية القيادة بواسطة STM32، وتم تحليل بروتوكولات الاتصال ثنائية الاتجاه بين الشبكة العصبية الاصطناعية القائمة على MCU ومحاكاة الكمبيوتر الشخصي.

تمكن محاكاة السيارة ذاتية القيادة بنجاح من قياس المسافات باستخدام مستشعرات الرادار وترجمة أوامر التوجيه إلى اتجاهات السيارة. تم إنشاء الاتصال بين نموذج الشبكة العصبية المنشور على وحدة التحكم الدقيقة STM32 ومحاكاة سيارة الكمبيوتر الشخصي، مما يسهل تبادل البيانات.

تتناول هذه الدراسة تحدي اختبار الأجهزة المتطورة في بيئات الوقت الفعلي وتؤكد على تبسيط الأنظمة الخوارزمية وجمع البيانات. كما تسلط الدراسة الضوء على أهمية تحويل البيانات بين التنسيقات الرقمية والتنظيرية من أجل اتصال سلس. يساهم إنجاز هذا العمل بنجاح في اختبار الأجهزة العصبية الاصطناعية الفيزيائية وبروتوكولات الاتصال الفعالة من أجل الاختبار في الوقت الفعلي.

الكلمات المفتاحية:

الخلايا العصبية الاصطناعية الفيزيائية، محاكاة السيارة ذاتية القيادة، بروتوكولات الاتصال، STM32، وحدة التحكم الدقيقة، الشبكة العصبية، بيئات الوقت الفعلي.

# Chapter 1: Introduction

## 1.1 Exploring Novel Approaches in the Electron Devices Department at BME

The ongoing research in the Electron Devices Department at BME delves into the realm of artificial intelligence architecture using vanadium oxide (VO2). This material, boasting a robust metal-insulator-transition (MIT) at approximately 341 K (68 °C) [3], holds immense promise for implementing an artificial neural network inspired by the intricate workings of the human brain.

In mirroring the operation of human neurons, where data is conveyed through both heat diffusion over short distances and electrical signals across longer distances, our pursuit aims to emulate this fascinating biological process. [7], [8]

The development of hardware-based artificial neuron devices has many technological challenges. Testing the VO2-based neuron network is enormously complex as test environment conditions can fluctuate, must be stable and reproducible teaching conditions, environment sensors deviation problems, etc.

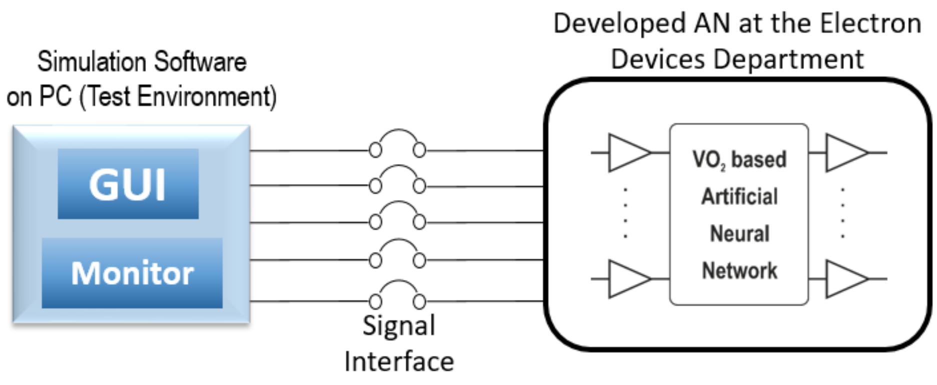

Developed AN at the Electron Devices Department
Simulation Software on PC (Test Environment)
GUI
Monitor
VO₂ based Artificial Neural Network
Signal Interface

**Figure 1. 1 Testing the developed AN with the test environment**

## 1.2 What is Artificial Neural Network (ANN)

Artificial Neural Networks (ANNs) stand as a paradigm for information processing inspired by the functioning of biological nervous systems, such as the human brain. These networks consist of fundamental processing units organized in multiple layers, mimicking the structure of the brain. Neurons, the basic processing units, undertake the tasks of input collection and output generation.

Biologically inspired, ANNs replicate the network of neurons in the human brain, where dendrites collect signals from neighbouring neurons in a confined space, and the cell body integrates and responds to these signals. The resulting response is then propagated through branching axons, interacting with the dendrite trees of multiple other neurons. [9]

### 1.2.1 ANN Biological inspiration

The human brain's intricate network comprises neurons connected to receptors (dendrites) and effectors (axons)[10], as shown in Figure 1. . It illustrates how dendrites gather signals, the cell body integrates them, and a response signal is distributed through axons, resembling the operation of artificial neural networks. [9] The researched VO2 material at the Department of Electron Devices is one of the best candidates to make a brain-like operation device. [8]

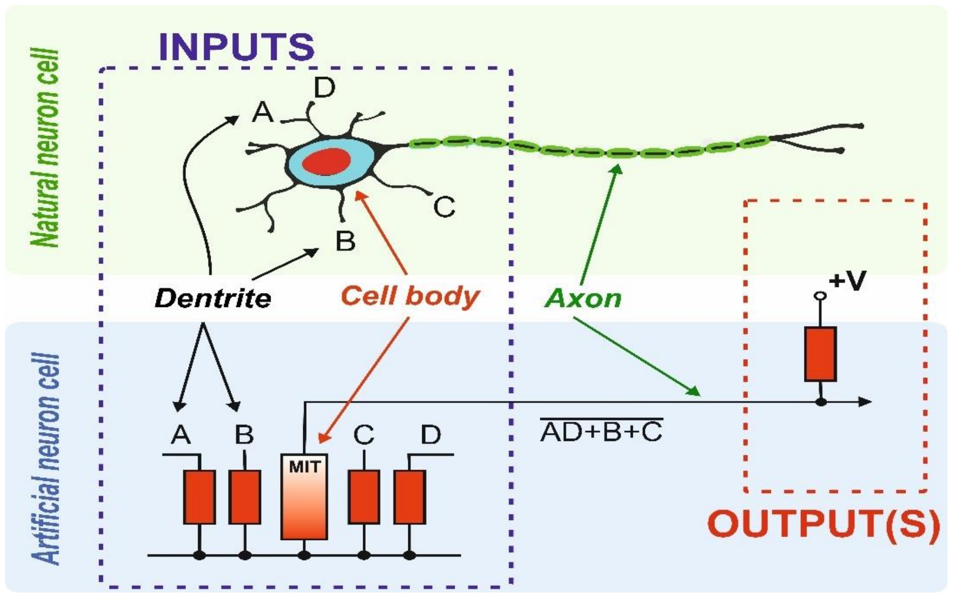


**Figure 1. 2 Biological Neuron**

### 1.2.2 ANN Methodology

ANNs serve as massively parallel computing models, simulating the intricate workings of the human brain. A multitude of basic processors interconnected via weighted connections forms the architecture of an ANN. Each unit, akin to a neuron, processes information locally, influenced by inputs received from other nodes. The neuron's output is a nonlinear function of its inputs, and weight modification through learning techniques plays a pivotal role in enhancing the network's capabilities.

The mathematical model of an ANN involves representing neuron connections as weights, where negative values denote inhibitory connections and positive values represent excitatory connections. The summation of inputs, modified by weights, is subjected to an activation function, determining the neuron's output magnitude. Typically, a range of output between 0 and 1, or -1 and 1, is considered acceptable. [11] Figure 1.3 provides a visual representation of the ANN mathematical model.

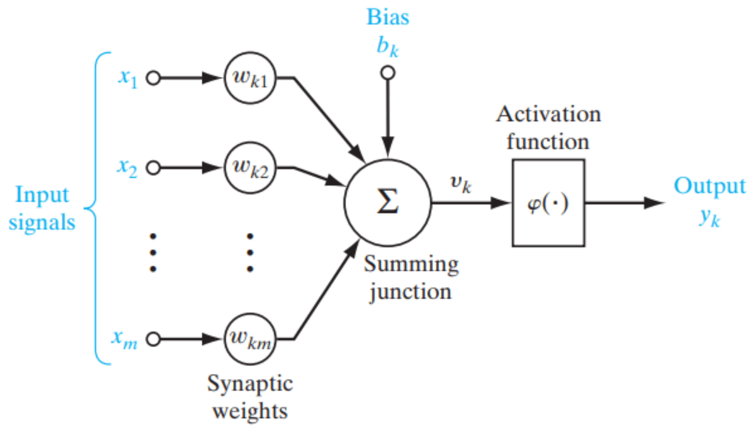


**Figure 1. 3 ANN Mathematical Model** [5].

From this model, the interval activity of the neuron can be represented as follows:

$$\text{Output} = \sum_{j=1}^{P} W_{kj} . X_j$$

**Equation 1. 1 Calculating Neuron output**

Equation 1.1 expresses the calculation of neuron output, where the output is determined by applying an activation function to the summation value.

## 1.3 Software Simulator for Hardware under development

As societal challenges grow in complexity, the need for innovative solutions becomes imperative. Computer simulation emerges as a powerful tool for analyzing, designing, and controlling complex systems. Engineers, system analysts, and researchers leverage simulation modelling to address real-world issues, offering a cost-effective and less disruptive alternative to experimentation with physical systems. [12]

## 1.4 TensorFlow and TensorFlow Lite

"TensorFlow is an end-to-end open-source platform for machine learning." [12] TensorFlow was developed by Google Brain Team developers as part of Google's Artificial Intelligence research group for machine learning and deep neural network research, although the technology is versatile enough to be applied in a variety of different fields. [13]

When TensorFlow is used to implement and train a machine learning algorithm (Object detection algorithm in our case), the development process ends with a model file that requires high resources like storage space or GPU to run inference. Such resources are not available on mobile devices (such as Android / iOS mobile phones or Raspberry Pi), which lead to the introduction of TensorFlow Lite as a solution to run machine learning models on mobile devices. TensorFlow Lite is a special aspect that is primarily intended for embedded devices such as mobile phones. For low execution delay and low load, a specialized memory allocator is used. TensorFlow Lite takes existing models and optimizes them in a ".tflite" file.[13]

The advantage of TensorFlow lite models is that they can be converted quickly and easily from TensorFlow models to mobile-friendly models. It also helps to build machine learning apps for iOS and Android devices easily. It facilitates offline inference on mobile devices, so it is a more effective alternative to server-based architectures for mobile model operation. TensorFlow Lite enables the execution of machine learning models on a smartphone, eliminating the need for an external API or server to do standard machine learning activities. As a result, the models will work even in case the device is not connected to the internet.[13]

The disadvantage of TensorFlow Lite is that it does not optimize the model in terms of size, which can lead to the need for external storage. The other disadvantage is that the precision of the model is sacrificed at the cost of reliability and optimization. TensorFlow Lite models, as a result, are less accurate than their full-featured counterparts. [13]

## 1.5 Related work

August 1991 and at the University of Stuttgart, Germany, A neural network simulator for Unix workstations Stuttgart Neural Network Simulator (SNNS) is described. The network simulation environment is a tool to generate, train, test, and visualize artificial neural networks. The simulator consists of three major components: a kernel operating on the internal representation of the neural networks and a graphical user interface based on X-Windows. [14]

Since 2014, Google has had a lot of internal projects for running neural networks that were just 14 kilobytes (KB) in size! They needed to be small because they ran on most Android phones' digital signal processors (DSPs), continuously listening for the "OK Google" wake words. These DSPs had only tens of kilobytes of RAM and flashed memory, and these specialized chips use only a few milliwatts (mW) of power. [15]

In 2015, a published publication through IEEE called Simulation of Mobile Robot Navigation Utilizing Reinforcement and Unsupervised Weightless Neural Network Learning Algorithm. The self-learning algorithm was embedded on a microcontroller and simulated in an open-source Simple 2-D Robot Simulator in Python PyGame simulator developed by M. Agapie. [16]

In 2018, the University of Guanajuato (Department of Electrical and Computer Engineering) Sergio Ledesma and Mario-Alberto implemented a Neural Lab simulator using object-oriented programming by a set of C++ classes. This simulator supports multi-layer feed-forward networks and probabilistic neural networks. [17]

IEMEK Journal of Embedded Systems and Applications published an article in 2020 called Pacman Game Reinforcement Learning Using Artificial Neural-network and Genetic Algorithm, and the purpose of this article is to reinforce the learning of the Pacman AI of the Simulator and utilize a genetic algorithm and artificial neural network as the method. In particular, building a low-power artificial neural network and applying it to a genetic algorithm was intended to increase the possibility of implementation in a low-power embedded system. [18]

In 2021 Eric Heiden and NeuralSim used a differentiable Simulator, providing an avenue for closing the sim-to-real gap. And find the simulation parameters that best fit the observed sensor readings connected to a Rasberry Pi and simulated on a PC. [19]

Uba. BV, in 2021, wrote in his thesis work about smart robot control information technology where methods and tools for the development of mapping system prototype held by using Arduino board and the mapping system prototype were drawn by PyGame library. [20]

Later in the same year, Alican Özeloğlu, İsmihan Gül Gürbüz, and İsmail San published a paper for implementing Deep reinforcement learning-based autonomous parking design with neural network compute accelerators where hardware on an FPGA and a simulation environment was created by using the Python-pygame module to implement the Deep Q-Learning method. [21]

One and half years ago, in March 2022, IEEE again published research by Dola Ram and Suraj Panwar on ”Hardware Accelerator for Capsule Network based on Reinforcement Learning”. The Hardware was FPGA, and the network was tested on a PyGame-based environment as stated in the research.

The exciting developments outlined above, ranging from the early Stuttgart Neural Network Simulator (SNNS) to recent innovations like the differentiable simulator by NeuralSim, represent pivotal milestones in the field. However, these are just a short introduction to the broader spectrum of contributions. Tools like NVIDIA CUDA for GPU acceleration and BindsNET for spiking neural networks facilitate simulation and significantly impact the broader domains of machine learning and artificial intelligence.

NVIDIA CUDA (Compute Unified Device Architecture) has revolutionized neural network simulation by harnessing the power of Graphics Processing Units (GPUs) for parallel computing. In 2006, CUDA enabled researchers to accelerate neural network training and simulation, significantly reducing computation time. Its impact on deep learning research and applications is substantial.

OpenAI Gym, an open-source toolkit, provides a diverse set of environments for developing and comparing reinforcement learning algorithms. Released in 2016, OpenAI Gym facilitates the simulation of neural networks in the context of reinforcement learning tasks. Its standardized interfaces and extensive set of environments have made it a valuable resource for researchers in the field [22]

BindsNET is a Python library designed explicitly for simulating spiking neural networks. Released in 2018, BindsNET allows researchers to model and simulate neural networks that closely mimic the spiking behaviour of biological neurons. This simulator is particularly valuable for exploring the dynamics of spiking neural networks and their applications in neuromorphic computing.[23]

## 1.6 Thesis objectives

In this Thesis work, as shown in Figure 1. 2, the diagram shows an implemented TensorFlow neural network deployed on an STM32 microcontroller. The NN is used to control a self-driving car simulator, which is running on a PC. The communication between the NN and the simulator is established through a USB serial port.

The NN is deployed on the STM32 microcontroller using a special library called TensorFlow Lite Micro (TFLM). TFLM is a lightweight version of the TensorFlow machine learning framework optimized for running on embedded devices.

The NN is trained on a dataset of distances measured between the centre of the car and the track borders using radars and directions where the car rotates left or right, or moves forward. The dataset is collected using a self-driving car prototype. The NN is trained to predict the direction value required to steer the car in the desired direction.

Once the NN is trained, it is deployed on the STM32 microcontroller. The microcontroller runs the ANN in real-time and predicts the direction required to steer the car. The direction is then sent to the self-driving car simulator through a UART communication protocol.

In the diagram, USB1 and USB2 are used as UART communication protocols to establish a P2P (Point-To-Point) connection between the STM32 microcontroller and the self-driving car simulator. This P2P connection sends and receives the steering wheel angle data.

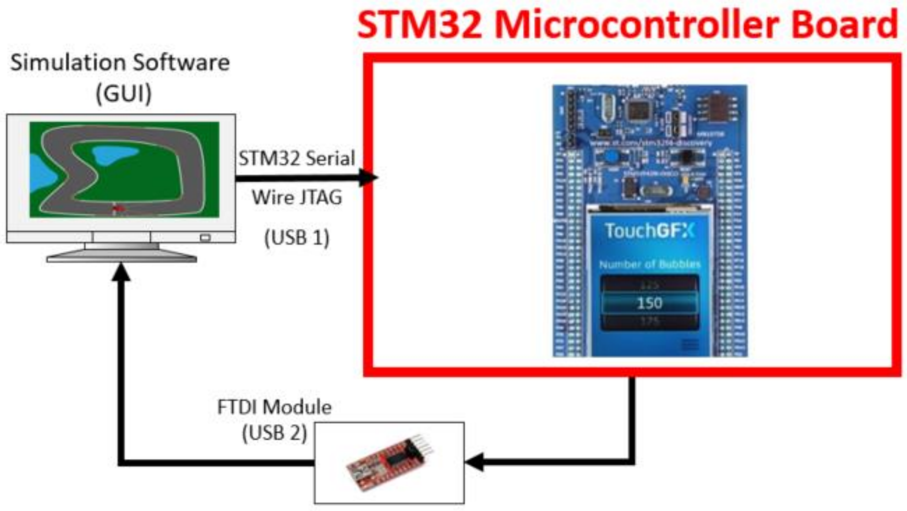

STM32 Microcontroller Board
Simulation Software (GUI)
STM32 Serial Wire JTAG (USB 1)
TouchGFX
FTDI Module (USB 2)

**Figure 1. 2 Thesis work objective**

# Chapter 2: PC Software Car Simulator Selection

A simulation model is a simplified representation of a system that improves my understanding, prediction, or even control of the system's behavior. A simulation system is "the collection of entities, such as devices or machines that act and interact together toward accomplishing some logical end in the simulation context”. For example, designing a mathematical-logical model of a real system and experimenting with this model on a computer. [24]

The system should be presented for simulation modelling in aspects that a computer performs. For example, if a set of variables can represent a system, modifying the value of the variables simulates the system moving from one state to another. Therefore, simulation involves tracking a model's changing dynamic behavior over time.

## 2.1 Open-source car simulator solutions

To train and evaluate my ANN hardware model, it is important to use an environment in where the simulator can learn and acquire the information needed for training.

Car simulators are the recent developments while training a car to run in an environment. They are very efficient in collecting huge amounts of data through their different built-in sensors, such as cameras, LIDAR, and RADAR sensors. Simulators provide real environments that contain town maps, racing tracks, and urban and rural environments. In addition, they provide many objects, such as pedestrians, bicycles and other driving vehicles of different shapes and types. Simulators can help cars learn to easily detect these kinds of objects, so they have become very popular to train autonomous cars to drive in different environments. There are many different simulators which are very powerful in autonomous car driving.

### 2.1.1 Three-Dimensional (3-D) car simulator

Many simulators are used in different scenarios and times, which are very powerful in autonomous car driving. CARLA and TORCS simulators are the two popular simulators. CARLA is an open-source autonomous driving simulator which gives a wide variety of working environments to train a car to drive in the urban environment. It gives many objects in the environment, such as traffic rules, pedestrians, and other different types of vehicles. The CARLA simulator is built on the client-server architecture. It is a more complex simulator for training as it requires higher computational capabilities. [25]

### 2.1.2 PyGame simulator

Simple DirectMedia Layer (SDL) is a multimedia library that allows access to hardware in a cross-platform fashion. The SDL library has Python bindings called the PyGame library. It consists of various top-level "parts", including Sprite, Surface, Font, Mixer, and others, each of which is required to produce a complete game. [26]

It offers mouse, keyboard, or joystick user input handling, game output via a screen for drawing shapes, rendering fonts, etc., and speakers for sound effects and music. Unfortunately, PyGame only works with 2-D graphics. ODE, or Open Dynamics Engine, is used here. The two main characteristics are:

Rigid Body Simulations are its initial and primary feature. PyODE simulates how opposing elements interact with different bodies. The physical characteristics of the bodies have little impact. However, physical characteristics like motion, speed, etc., are important. To use this library, we first create a Car body and configure its properties, such as rotation, motion, and speed. Then, the library gives us a few helpful calls to make objects of various forms by defining their parameters (for example, making a circle by defining its radius or making a line by defining two points). Finally, the car's radar will be drawn using shapes (discussed in Chapter 4 in detail). [26]

The second characteristic of PyODE is that it assists with determining when two objects collide. This feature is very important to my Car game simulator. This feature is used to keep the car on track without sliding away from the road. And detect when the car colloids the edge of the road.

### 2.1.3 PyGame Advantage over 3-D car simulation

- Python is an easily used programming language (High-Level Language).
- PyGame library provides all the functions required for implementing a self-driving car simulator.
- Made physics car game easier to implement.
- All of this was possible when using PyODE properly.
- Proper simulation of a rigid body (Car).

### 2.1.4 PyGame drawbacks:

- Basics laws of physics are not very easy to implement.
- One fault in the code loop of the game will disturb the whole system.
- Physics simulation may not be very accurate.

## 2.2 Open-source Car Simulator Solutions Results

Since this thesis work does not involve 3-D graphics, PyGame's restriction does not affect it. Therefore, it is undoubtedly the best game creation library for my system, thanks to its controller compatibility and in-game physics engine PyODE.

After choosing a game development library, let us investigate the parameters required to create physics-aware games with these libraries.

Finding the distance between a moving object (car) and an obstacle that allows the car to safely react by steering away from the object, slowing down, or perhaps stopping altogether is necessary for collision-free navigation. The distance sensor must therefore have a high degree of accuracy, sensitivity, and precision to estimate the distance. Market alternatives vary, as shown in Table 2. 1, which includes Time of Flight (TOF), IR proximity, ultrasonic distance, laser, and Camera. They can all determine an object's proximity without making physical contact.

The radars drawn in the designed car simulator are a simulated version of the infrared radar sensors. Which serves the purpose of measuring the distance from the car to the track's edge, providing data utilized as input for the neural network.

| | Ultrasound | Infrared | Laser | Time-of-Flight |
|---|---|---|---|---|
| High reading frequency | | | √ | √ |
| Long range | | | √ | √ |
| Small form factor | √ | √ | | √ |
| Eye safety | √ | √ | Class 1 only | √ |
| Use multiple sensors. | | | | √ |

**Table 2. 1 Comparison between different types of sensors** [21]**.**

# Chapter 3: Hardware

The first task, is to Communicate between two computers using STM32 MCU as a communication interface and UART protocol for communication. This task aims to investigate the possibility of controlling the car simulator from a remote device that hosts the self-driving algorithm. The hosting device can send commands (directions) to the client device hosting the software car simulator.

The second task is to complete the testing of the communication between the physical neural network ANNs (TFLM model deployed to STM32 microcontroller board) and the test environment which involves a seamless flow of signals. DAC and ADC play a crucial role in ensuring that signals are accurately transmitted between the neural network and the external environment. One way to test physical neural network peripherals is to connect them to a test environment through a digital-to-analog converter (DAC) and an analogue-to-digital converter (ADC). This allows the neural network to interact with the real world and receive feedback on its performance.

The third task is deploying a self-driving car algorithm on a microcontroller to control a car simulator optimize the algorithm for deployment on a microcontroller, and ensure that it can make accurate decisions in real-time based on the data it receives from the car simulator's sensors.

In this chapter, the hardware structure will be shown, and the technical specifications of the modules will be summarized. The main components are the STM32F429IDISCOVERY, USB-TTL (FT232RL) converter module.

## 3.1 STM32 Microcontroller Unit (MCU)

As the AN hardware is still under development, therefore, the STM32 microcontroller unit (MCU) based on the Arm Cortex-M processor with a deployed AN deployed is used for simulating the AN network.

### 3.1.1 Introduction to STM32F429I-DISCOVERY

The STM32F429IDISCOVERY shown in **Error! Reference source not found.** is a development board based on the STM32F429ZIT6 microcontroller. It offers a comprehensive set of peripherals and connectivity options to support the development of embedded applications. One of the most significant features of the 32F429IDISCOVERY board is its support for the X-Cube-AI package, which provides artificial intelligence capabilities for edge devices. [27]

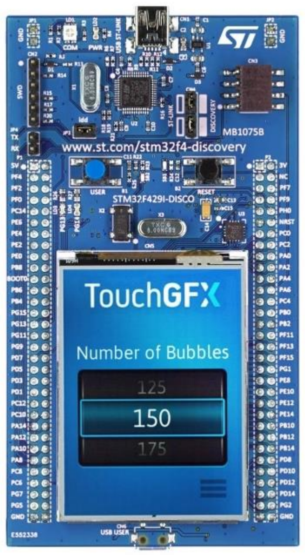


**Figure 3. 5 32F429IDISCOVERY Board**

#### 3.1.1.1 STM32F429IDISCOVERY RTC (Real-Time Clock) and Backup Registers:

The STM32F429IDISCOVERY board has a real-time clock (RTC) module that provides accurate timekeeping even when the system is powered off. In addition, the RTC module includes a battery backup circuit that ensures the RTC continues to operate even when the main power source is disconnected.

The board also has backup registers that store critical system information such as configuration settings, calibration data, and security keys. The RTC battery backup circuit powers the backup registers, ensuring that the data remains intact even during power outages. [27]

#### 3.1.1.2 Serial Wire JTAG (Joint Test Action Group) Debug Port (SWJ-DP):

The STM32F429IDISCOVERY board includes a Serial Wire JTAG (SWJ-DP) interface for debugging and programming the microcontroller. [27]

#### 3.1.1.3 Universal Synchronous/Asynchronous Receiver Transmitter (USART):

The STM32F429IDISCOVERY board has four Universal Synchronous/Asynchronous Receiver Transmitter (USART) interfaces that can communicate with external devices using serial communication protocols such as UART, SPI, and I2C. The USART interfaces support a wide range of baud rates and data formats, making them suitable for a variety of applications. [27]

## 3.2 Communication between two computers using STM32 with USB-TTL module.

The FT232L-M shown in Figure 3.  is a printed circuit board (PCB) that converts USB to Time-To-Live (TTL) serial UART. This USB developed by FTDI (Future Technology Devices International) is a serial UART interface Integrated Circuit (IC), which manages USB signaling and protocols. The PCB is a quick and easy method of joining TTL-level serial-interface devices to USB.

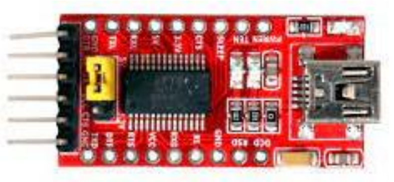


**Figure 3. 1 USB-TTL Module**

## 3.3 Analysis of Understanding STM32 UART onboard communication protocol

Different communication protocols, including I2C (Inter-Integrated Circuit), SPI (Serial Peripheral Interface), and UART, are supported by both high-level and low-level processors (Universal Asynchronous Receiver-Transmitter). While UART is asynchronous, I2C and SPI are both synchronous (operated by a clock) and do not need a clock [28]. Since the STM32 is used to represent the ANN device to be tested in future. Therefore, given that UART is a device-to-device protocol, it was decided that it is a good option for fulfilling the requirements. In serial transmission, data is sent over a single line or wire bit by bit. Therefore, two wires are required for successful serial data transfer in two-way communication. Depending on the application and system requirements, serial communications frequently require less wiring and circuitry, which lowers the implementation cost. [28]

UART is a hardware communication protocol that uses asynchronous serial communication with a programmable speed. Asynchronous refers to the absence of a clock signal that would have allowed the output bits from the transmitting device to be synchronized with those at the receiving end. Both devices are connected, as shown in Figure 3.2, with the transmitter (Tx) of UART1 connected to the receiver (Rx) of UART2 and the receiver (Rx) of UART1 to the transmitter (Tx) of UART2. The connection of the two wires creates a two-way communication channel [30].

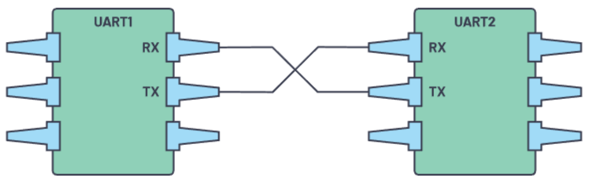


**Figure 3. 2 Two UARTs directly communicate with each other** [31]**.**

The advantages of using UART are that they simply operating and widely used, it is well documented with many resources, no clock needed and parity bit to allow for error checking.

Disadvantages of using UART are that they limited the size of the data frames up to only 9 bits, limited multiple uses of master systems and enslaved persons. Low data transmission.

Despite the disadvantages of the UART protocol, these disadvantages will not impact my work. Because the data required to be transmitted is less than 9 bits, on the other hand, it was used only one master and one slave device in this system for each UART channel. So, UART is the optimal solution for this work regarding the advantages and disadvantages.

## 3.4 Circular Buffer

A circular buffer, or a circular queue, is a data structure that efficiently manages a fixed-size buffer. When transferring successive data values from a producer thread to a consumer thread, a circular buffer is a useful utility that allows the data to be retrieved in a FIFO (First In, First Out) order [32].

In a circular buffer, data is stored in a block of memory, and two pointers, known as the head and tail, are used to keep track of the buffer's current position. As shown in figure 3.3 the head pointer points to the next available location for writing new data, and the tail pointer points to the oldest unread data in the buffer [33].

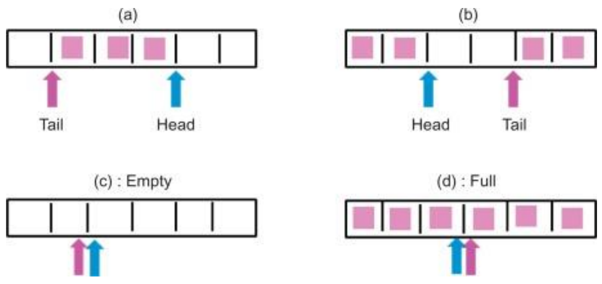


**Figure 3. 3 Circular Buffer Head and Tail** [33]

## 3.5 Schematic diagram of hardware circuit used to study Communication.

As shown in Figure 1.3 in the introduction chapter, the TensorFlow neural network deployed on an STM32 microcontroller is used to control a self-driving car simulator running on a PC. The communication between the NN and the simulator is established through a USB serial port.

The following are the main components of the system:

- STM32 microcontroller: the STM32 microcontroller is a powerful and versatile microcontroller well-suited for running ANN algorithms.
- Analog to digital converter (ADC): the ADC converts the analogue signals from the radars to digital signals that can be processed by the microcontroller.
- Digital to analogue converter (DAC): the DAC converts the digital output signals from the microcontroller to analogue signals that can be used to control the actuators.
- FTDI module: the FTDI module provides a USB interface to the microcontroller board. This allows the microcontroller board to be programmed and debugged from a computer.
- Simulation software: the simulation software can be used to simulate the behaviour of the ANN before it is deployed on the microcontroller board.
- Physical ANN: the physical ANN is the hardware implementation of the ANN algorithm. It consists of a network of interconnected neurons.

The following steps describe how to connect a physical neural network to a test environment through a DAC (Digital-to-Analogue) and ADC (Analogue-to-Digital), as shown in Figure 3.4:

- Identify the input and output signals of the neural network. The input signals are the data the neural network will be trained on, and the output signals are the results of the neural network's calculations.
- The input signals of the neural network to the DAC. The DAC will convert the digital input signals to analogue signals that can be understood by the physical neural network peripherals that will be tested.
- Connect the output signals of the DAC to the ADC. The ADC will convert the analogue input signals to digital signals again as a digital input signal to the neural network that the neural network can understand.

- Connect the DAC and ADC to the test environment. The test environment will provide the input signals to the neural network and receive the output signals from the neural network.

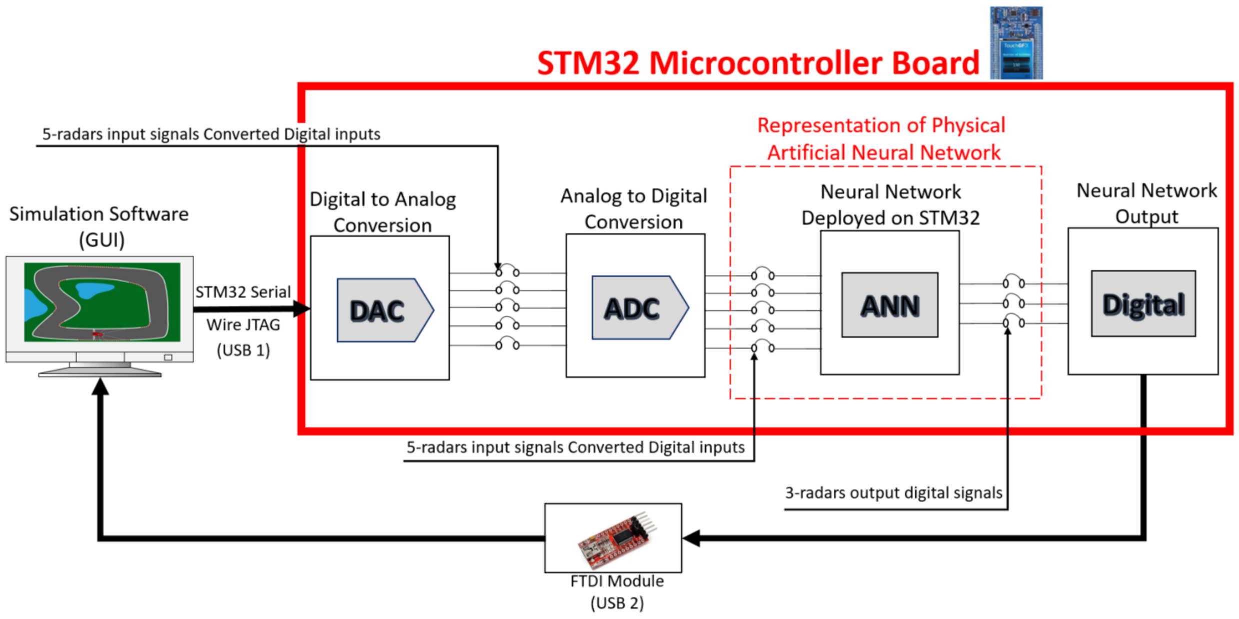

STM32 Microcontroller Board
Representation of Physical Artificial Neural Network
5-radars input signals Converted Digital inputs
Simulation Software (GUI)
Digital to Analog Conversion
DAC
Analog to Digital Conversion
ADC
Neural Network Deployed on STM32
ANN
Neural Network Output
Digital
STM32 Serial Wire JTAG (USB 1)
5-radars input signals Converted Digital inputs
3-radars output digital signals
FTDI Module (USB 2)

**Figure 3. 4 Hardware Circuit Diagram**

Connecting a physical neural network to a test environment through a DAC and ADC is a straightforward process. By following the steps outlined above, anybody can quickly and easily connect their neural network to a variety of test environments, including simulation software and physical systems.

This connection could be used to test the neural network in a variety of scenarios and collect data on its performance. This data can then be used to improve the performance of the neural network.

Here are some additional considerations for connecting a physical neural network to a test environment:

- Choose the right DAC and ADC: the DAC and ADC should have a sufficient resolution and sampling rate to meet the needs of your neural network.
- Calibrate the DAC and ADC: it is important to calibrate the DAC and ADC to ensure that convert signals accurately.
- Use a noise filter: The DAC and ADC may introduce noise into the signal. It is important to use a noise filter to reduce the noise level.
- Test the connection: once the DAC and ADC are connected, it is important to test the connection to ensure it works properly.

### 3.5.1 Choose the DAC and ADC

In the implementation of our neural network on the STM32 microcontroller, selecting the appropriate Digital-to-Analog Converter (DAC) and Analog-to-Digital Converter (ADC) is critical for ensuring accurate signal conversion. The choice of these components directly impacts the performance and reliability of the neural network.

**DAC Selection (Digital-to-Analog Converter) [29]**

The DAC should possess to first sufficient Resolution to ensure that the DAC resolution aligns with the precision required by the neural network. Higher resolution contributes to finer control over analogue output signals.

Second the compatibility where to confirm that the chosen DAC interfaces seamlessly with the microcontroller, meeting electrical and protocol specifications.

**ADC Selection (Analog-to-Digital Converter) [29]**

Consider the following factors when selecting an ADC. First, resolution and sampling rate where adequate resolution and sampling rate are essential for accurately capturing analogue signals. This is particularly crucial for preserving the fidelity of neural network computations.

Second, similar to the DAC, the ADC should be compatible with the microcontroller and capable of handling the expected input signal levels.

### 3.5.2 Calibrate the DAC and ADC

To ensure the accuracy of signal conversion, calibrating both the DAC and ADC is a crucial step. Calibration involves aligning the converters' output with the expected values. This process enhances the reliability of the neural network's interactions with the physical environment. [29]

**Calibration Steps:**

- Define Reference Signals: establish known reference signals that can be used to validate the accuracy of the converters.
- Adjust Gain and Offset: fine-tune the gain and offset settings of both the DAC and ADC to minimize any discrepancies between the actual and expected signal values.
- Verification: validate the calibration by comparing the output signals with the expected values under various input conditions.

### 3.5.3 Use a noise filter

In real-world applications, DAC and ADC operations may introduce unwanted noise into the signal. Implementing a noise filter is essential to mitigate this potential interference. [29]

**Noise Filter Considerations:**

- Filter Type: choose an appropriate filter type (e.g., low-pass filter) based on the characteristics of the noise present in the system.
- Cutoff Frequency: Set the cutoff frequency of the filter to attenuate high-frequency noise while allowing the essential components of the signal to pass through.

### 3.5.4 Test the connection

After configuring the DAC and ADC, thorough testing is necessary to validate the integrity of the connection between the physical neural network and the test environment. [30]

**Testing Procedures:**

- Input Signal Verification: confirm that the DAC accurately converts digital input signals to analogue signals.
- Output Signal Verification: validate that the ADC precisely converts analogue output signals back to digital format.
- Dynamic Testing: test the connection under dynamic conditions to simulate real-world scenarios.
- Fault Tolerance: assess the connection's robustness by introducing controlled faults and observing the system's response.

# Chapter 4: Software

## 4.1 Integrated Development Environment (IDE)

### 4.1.1 PyCharm IDE

A large number of expert developers across industries use PyCharm as their preferred Python Integrated Development Environment (IDE). It is thought to be the best choice for Python developers and is created by the Czech business JetBrains. Its cross-platform compatibility, which enables developers to use it on other operating systems, is one of its primary advantages. PyCharm's support for parallel Python script execution is one of its strongest features. [31]

### 4.1.2 STM32CubeIDE

STM32CubeIDE, the official ST software IDE, is a powerful C/C++ development platform designed for working with STM32 microcontrollers and microprocessors. This integrated development environment offers a range of essential features, including peripheral configuration, code generation, code compilation, and debugging capabilities. Built on the Eclipse framework and utilizing the GCC toolchain, STM32CubeIDE provides a robust and efficient environment for software developers working on STM32-based projects. It streamlines the development process and empowers programmers to create and debug their applications effectively. [32]

## 4.2 GUI game graphics by PyGame library [33]

### 4.2.1 Create a game, display car, and driving functions, and adjust user inputs.

The initial step in crafting the Graphical Car Simulator (see Appendix A) involves the creation of a window featuring a track. The objective is to enable the car to move forward upon pressing the arrow key. Commence by importing essential modules such as pygame, os, math, sys, and neat libraries.

Following this, establish the dimensions of the screen window (see Appendix A). The width is set at 1244 pixels, aligning with the track image's width (imported as a .png file), while the height stands at 1016 pixels, matching the track image's height. Initialize the game screen and load the track image as "*track.png*" serving as the default track for both testing and training the ML model.

Craft a car class derived from the sprite class, renowned for handling visible game objects like the car (see Appendix A). In the "*init*" function, initialize the parent class, set the car's image, and initiate the drive state variable. Specify the image's rectangle, integrate a velocity vector for speed, establish the car's angle at zero, and introduce a rotational velocity variable for subsequent use. Also, include a "*direction*" variable to facilitate steering.

The "*update*" function is designated for updating the car's status, enabling it to both drive and rotate (see Appendix A). The *'drive'* function involves incrementing the car's coordinates if the drive state is true, mimicking the car's movement across the screen. The 'rotate' function involves scaling down the original image. If the direction is 1 or -1, it adjusts the angle and velocity vector accordingly.

Create a container employing the single group function to hold a single sprite image (see Appendix A). Add a sprite object as an instance of the car class.

Proceed to set up the main loop within the "*eval_genomes*" function to assess the fitness of cars navigating the track (see Appendix A). Initiate a variable *'run'* set to true and implement a while loop contingent upon 'run' retaining its true state. Integrate standard code to exit the game upon pressing the close button, blit the track image, and manage key inputs to determine the drive state and direction.

Within the main loop, depict the car on the screen, update its status, refresh the display, and invoke the "*eval_genomes*" function (see Appendix A).

This methodical approach ensures a coherent progression in developing a Graphical Car Simulator with user engagement and seamless car movement.

### 4.2.2 Adding Radars and Collision Points

The radars serve the purpose of measuring the distance from the car to the track's edge, providing data utilized as input for the neural network. To achieve this, a function call to the "*radar*" function is created within the update function of the car class. Subsequently, a new function, "*radar*," is established. Within this function, a variable named "*length*" is initiated and set to 0, signifying its role in determining the radar's length (see Appendix A). The radars commence from the car's centre with a length of 0 and progressively extend until reaching the track's edge.

Following this, the x and y coordinates to the car's centre are defined, and a while loop is introduced. This loop extends the radar ray until it encounters the grassy area, incrementing the length by 1 in each iteration. The endpoint of the radar is then calculated. Visual representation is provided as the radar is drawn on the screen, featuring a straight line connecting the car's centre to the radar's endpoint as its primary component. Additionally, a green dot is placed at the top of the radar.

Recognizing that a single radar might not suffice as input for the neural network, five radars are created using a for loop. This loop iterates through five different radar angles, and the "*radar*" function is called with the radar angle as an argument.

The collision mechanism of the game is established to detect when the car leaves the track and ventures onto the grass. Within the "*init*" function of the car class, an "*alive*" variable is introduced, initially set to true when the car starts on the track. However, when the car leaves the track and enters the grass, it transitions to false. The call function and the collision function are added to the update function.

The "*collision*" function is defined further below. A "*length*" variable is initialized to 40, representing the distance between the car's centre and the collision point (see Appendix A). Two collision points are created at the right and left headlights of the car. An if statement checks the color underneath these collision points. If the color under either point is identified as green, indicating that the car is on grass, the "*alive*" variable is set to false. Finally, the collision points on the screen are visually depicted.

## 4.3 Implementing Classification Model for Self-Driving Cars using Sequential Keras Model

In our earlier discussions, it was explored the implementation of machine learning models on the STM32 MCU. It was found that the optimal approach for executing the classification model would involve converting it to the TFLite format. TFLite, tailored for mobile applications and embedded devices, emerges as the preferred choice for running the model on the hardware of the self-driving car.

### 4.3.1 Data collection:

The radars play a crucial role in gauging the distance between the car and the track's edge. A list named "*Input*" is generated by the "*data()*" function, consisting of five zeros.

Subsequently, a loop is initiated to populate the input list with data from the five radars, each providing the distance from the car's centre to the radar tip (see Appendix A).

An if statement is employed to deactivate the drive state "0" when no keys are pressed. Another if statement is formulated, activating the drive state when the up key on the keyboard is pressed, yielding "0" in the absence of right or left key inputs. To manage user inputs for steering, the variable "*direction*" is set to "1" when the right key is pressed and "-1" when the left key is detected.

Ultimately, data from both radar and direction inputs is gathered from the car simulator and archived in an Excel sheet. The objective is to train a classification model based on the radar data to predict whether the car should turn right or left. Subsequently, the TensorFlow model is converted to the TFLite format, facilitating its operation on the STM32 MCU.

### 4.3.2 Data Preprocessing:

Before training the classification model, first preprocessing the data, involves cleaning data, dropping Na (Not Applicable) values, clearing duplicates and splitting it into training and testing sets.

The "*to_categorical*" function from Keras is used to encode the target variable y, which is a categorical variable, into one-hot encoded vectors. The "*to_categorical*" function is applied to convert the output variables of the directions into its encoded form (see Appendix B).

Using the "*to_categorical*" classification task is particularly useful for categorizing different output directions. Where the three output classes represent different directions "-1" for right, "-1" for left, and "0" for straight. The "*to_categorical*" function will transform these categorical labels into one-hot encoded vectors, where each direction corresponds to a unique 3-bit binary representation.

Finally, the "*train_test_split()*" function from scikit-learn is used to split the data into training and testing sets. The features are stored in the variable X, and the encoded target variable is stored in Y (see Appendix B). The "*test_size()*" parameter is set to 0.2, meaning 20% of the data will be used for testing. The resulting split data is stored in "*X_train*", "*X_test*", "*y_train*", and "*y_test*". These variables are used for training and evaluating a classification model.

### 4.3.3 Data define the network, and train the Classification model:

The first step in building a neural network is to define the architecture of the model. In this step, a sequential model is used using the "*Sequential()*" function, which allows us to add layers to the model in sequential order (see Appendix B).

Four dense layers are defined, with the first layer having 16 nodes and an input shape of 5 (5 radar values). The next two layers have 32 nodes, and 16 nodes with a 20% dropout respectively, and the activation function used for these layers is "*relu*". Finally, the output layer is defined with 16 nodes, representing the 3 classes of the classification problem (Straight, Right and Left), and the activation function used is "*softmax*", which is commonly used in multi-class classification problems. Then the model is compiled with "*Mean-Square Error*" (MSE) as the loss function and accuracy as the evaluation metric. The Adam optimizer is used with a learning rate of 0.001 (see Appendix B).

Finally, the model is trained on the training data, specifying a batch size of 32 and training the model for 50 epochs, and a validation split of 0.2, meaning that 20% of the training data is used for validation during training (see Appendix B).

### 4.3.4 Evaluating the Classification Model:

One important aspect of evaluating a machine learning model is visualizing its training and validation performance. The model is evaluated using evaluation metrics like loss, precision-recall, and f1-score (see Appendix B).

In addition to these techniques, it is also important to analyze the model's architecture and parameters. The model's architecture refers to the structure of the network, including the number of layers, the number of nodes in each layer, and the activation functions used in each layer. The model's parameters refer to the values of the weights and biases that are learned during training.

### 4.3.5 Saving model to disk, and converting implemented TensorFlow model to TensorFlow Lite model:

The first step involves saving the trained model in JSON format using the "*to_json*" method and saving the trained model's weights in HDF5 format using the "*save_weights*" method (see Appendix B).

The "*TFLiteConverter*" class is used to convert the Keras TensorFlow model to a TensorFlow Lite model, where the converted TensorFlow Lite model is saved in a binary format with the "*.tflite*" suffix.

## 4.4 Deploying AI model on STM32F429I-DISCOVERY and testing deployed model on GUI python PyGame

Once the model is trained, quantized, and converted to a TFLite file. The next step is testing inference by giving it inputs and comparing outputs to the model outputs. Netron software can be used to visualize the Keras and TFLite model files to help verify the input and output formats, along with how the layers are connected. In Figure 4. 1 Model visualization using Netron software the model used in this work is visualized.

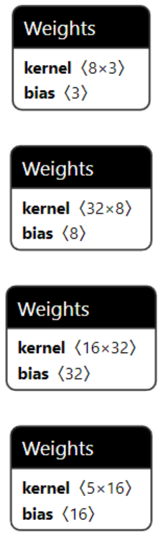
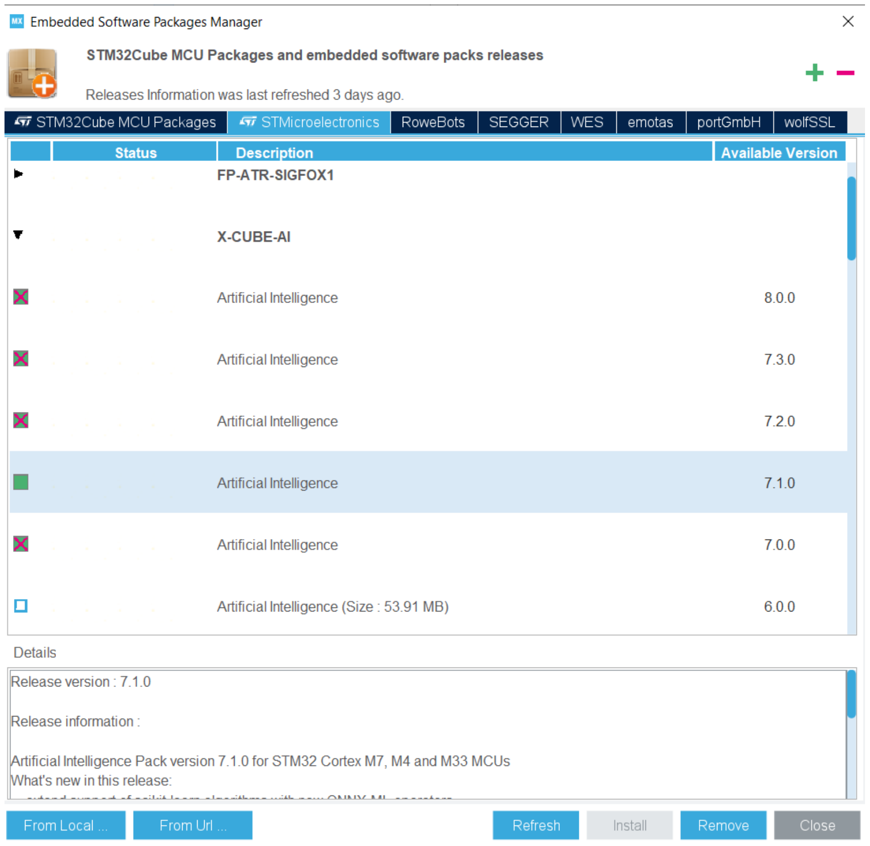


**Figure 4. 1 Model visualization using Netron software**

### 4.4.1 STM32F429I-DISCOVERY C code implementation using STM32CubeIDE.

In STM32CubeIDE, as a first step, the "*help*" menu is accessed, followed by selecting "*manage embedded software packages.*" The X-Cube-AI package is installed, as illustrated in Figure 4.2.


Embedded Software Packages Manager

STM32Cube MCU Packages and embedded software packs releases

Releases Information was last refreshed 3 days ago.

| | Status | Description | Available Version |
|---|---|---|---|
| ▶ | | FP-ATR-SIGFOX1 | |
| ▼ | | X-CUBE-AI | |
| ☒ | | Artificial Intelligence | 8.0.0 |
| ☒ | | Artificial Intelligence | 7.3.0 |
| ☒ | | Artificial Intelligence | 7.2.0 |
| 🟩 | | Artificial Intelligence | 7.1.0 |
| ☒ | | Artificial Intelligence | 7.0.0 |
| ☐ | | Artificial Intelligence (Size : 53.91 MB) | 6.0.0 |

Details

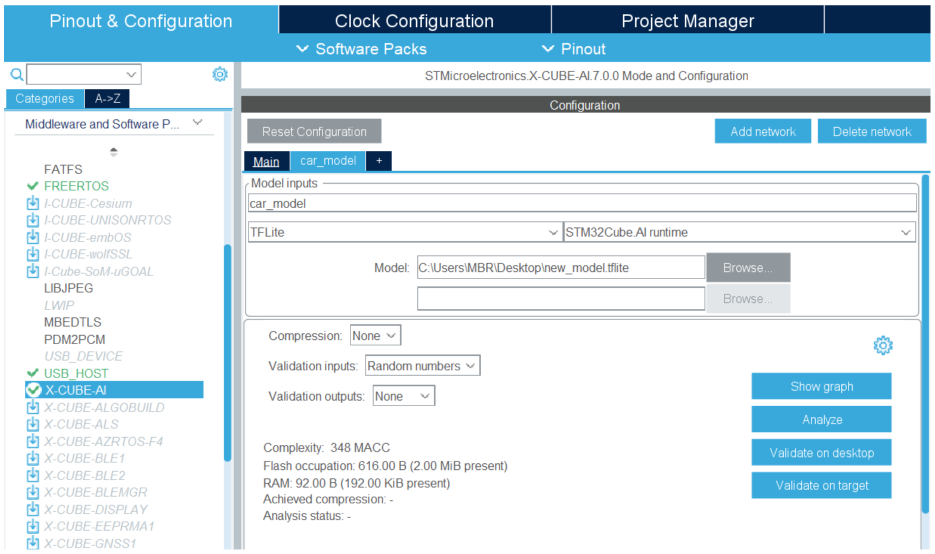
Release version : 7.1.0

Release information :

Artificial Intelligence Pack version 7.1.0 for STM32 Cortex M7, M4 and M33 MCUs
What's new in this release:

**Figure 4. 2 Install X-Cube_AI Library**

Next, the board selector tab is navigated, and the STM32F429I-DISCOVERY board, supporting X-Cube-AI in this context, is chosen. Subsequently, a network is added, opting for TFLite. The process involves saving and generating code, resulting in the addition of an X-Cube-AI directory to the project within the app folder.



STMicroelectronics.X-CUBE-AI.7.0.0 Mode and Configuration

Configuration

Model inputs
car_model
TFLite | STM32Cube.AI runtime
Model: C:\Users\MBR\Desktop\new_model.tflite

Compression: None
Validation inputs: Random numbers
Validation outputs: None

Complexity: 348 MACC
Flash occupation: 616.00 B (2.00 MiB present)
RAM: 92.00 B (192.00 KiB present)
Achieved compression: -
Analysis status: -

**Figure 4. 3 Configure TFLite model in the STM32CubeIDE**

Within the generated files, specifically in the source code "*car_model.c*" essential functions for initializing and running the model inference are present. Additionally, in the "*car_model_data.c*" source code, a neural network representation is depicted in the form of a substantial array.

In the "*main.c*" modifications are made to the main code. Initially, the C standard input-output library is included, followed by the inclusion of AI-defined platform headers containing necessary functions. Subsequently, the model files are included. Variables are then declared, encompassing a buffer for serial output strings, error codes, and a timestamp.

A memory buffer is created to store intermediate calculations for the neural network. Input and output tensor arrays are generated for this specific model. A pointer is established for the model, along with several structs pointing to input and output data and associated metadata. A parameter struct pointing to the model and working memory is also created.

Configuring the wrapper structs enables running inference multiple times with different data in a single iteration. Additionally, a data pointer is formed to point to input and output buffers. An instance of the neural network is generated using the *'ai_car_model_create()'* function, and the model is initialized with the init function.

The while (1) loop commences by awaiting a start byte (0x01) on the UART line, achieved through the "*HAL_UART_Receive()*" function (see Appendix C). Once the start byte is received, the loop enters another while (1) loop, reads a block of data from the UART buffer until a stop byte (0x02) is received. The received bytes are stored in a buffer named *"buffer"* with a maximum data size of 10. If the number of bits exceeds 10, the loop is exited. Otherwise, it continues until the stop-byte (0x01) is received (see Appendix C).

Subsequently, the buffer array is transformed into an array of integers called an array. This array is then utilized as input for the neural network model. Following that, the input buffer is filled with the input data. For testing purposes, the current timestamp is obtained from the timer. An inference is then performed with the model using the "*ai_car_model_run*" function, providing it with the input and output buffers along with the model (see Appendix C).

### 4.4.2 GUI Python code modifications to communicate with the deployed model on STM32F429I-DISCOVERY

In this part, the GUI python code is modified to send the radar values of the car position to the STM32 board through a UART line and receive back the predicted direction from the AI model deployed on the board.

First, the *"eval_genomes()"* function is modified to set up a connection on UART serial port "*COM6*" with a baud rate of 115200 (see Appendix D). Then it enters a while true loop, which runs continuously.

The radar array data is set up in the form of five integers based on the current state position of the car and packs each integer into two bytes in big-endian format. The resulting list of bytes is then sent over the serial port to the external device, along with start and end bits

After sending the data, the GUI waits for a response from the deployed model on the STM32. The response is a single byte representing the predicted direction of the car (0, 1 or 2)

The program then updates the state of the car based on the response received from the external device. If the response is 0, the car moves to the left. If the response is "1", the car moves to the right. Lastly, if the response is "2" the car continues to move in straight direction.

# Chapter 5: Results and Evaluation

## 5.1 PyGame 2-D car simulator building result.

As mentioned in the previous experimental work, radars are managed to be drawn, which will be used to calculate the distance between the car and the grass area around the track. Since one single radar is not going to be enough of an input for my neural network, making multiple five radars will be satisfying, as shown in Figure 5. . These radars help in measuring how far the car is away from the grassy area of the track in all directions as the car moves forward.

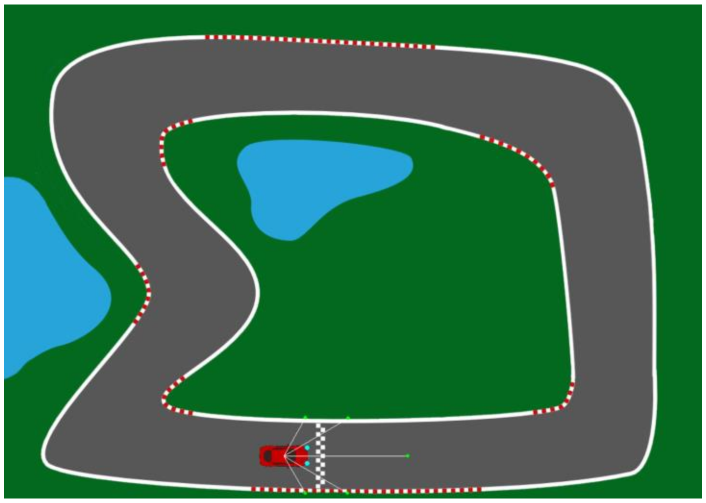
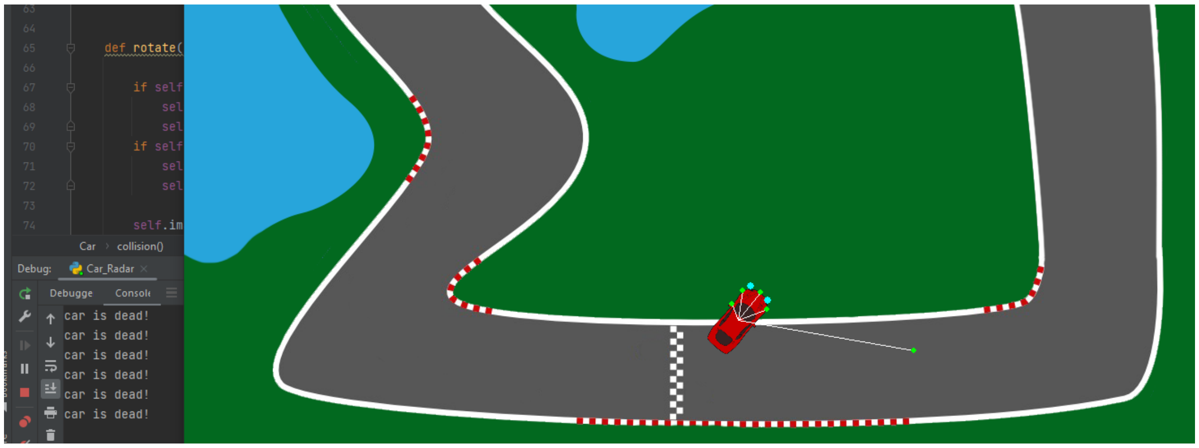


**Figure 5. 1 Car Simulator – Radar Drawing**

To show that the collision function does work. A print statement is added within the if statement, which is going to print "car is dead!" whenever the car crosses into the green area of the map, so now when running the game again, drive the car right into the grass. It is observed that in the console output, it logs that the "car is dead!" as shown in Figure 5. .



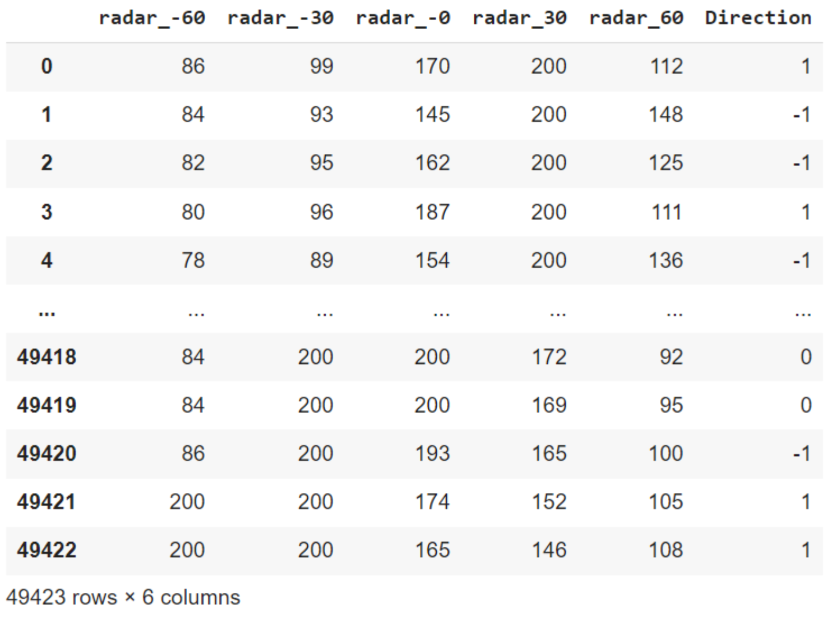
```
    def rotate(

        if self
            sel
            sel
        if self
            sel
            sel

        self.im
```

```
car is dead!
car is dead!
car is dead!
car is dead!
car is dead!
car is dead!
```

**Figure 5. 2 Car Simulator – Collision detection**

## 5.2 Implementing Classification Model for Self-Driving Cars using Sequential Keras Model

The resulting data table was collected from a car driven on a track in this part; it is discussed and analyzed. The data table has six columns and almost 49,500 rows, as shown in Figure 5. . The columns are "radar_-60", "radar_-30", "radar_-0", "radar_30", "radar_60", and direction. The first five columns represent the radar readings from different car angles. The last column, "Direction", represents the three classes in which the car turned "-1" for left, "1" for right, and "0" for Straight.



| | radar_-60 | radar_-30 | radar_-0 | radar_30 | radar_60 | Direction |
|---|---|---|---|---|---|---|
| **0** | 86 | 99 | 170 | 200 | 112 | 1 |
| **1** | 84 | 93 | 145 | 200 | 148 | -1 |
| **2** | 82 | 95 | 162 | 200 | 125 | -1 |
| **3** | 80 | 96 | 187 | 200 | 111 | 1 |
| **4** | 78 | 89 | 154 | 200 | 136 | -1 |
| **...** | ... | ... | ... | ... | ... | ... |
| **49418** | 84 | 200 | 200 | 172 | 92 | 0 |
| **49419** | 84 | 200 | 200 | 169 | 95 | 0 |
| **49420** | 86 | 200 | 193 | 165 | 100 | -1 |
| **49421** | 200 | 200 | 174 | 152 | 105 | 1 |
| **49422** | 200 | 200 | 165 | 146 | 108 | 1 |

49423 rows × 6 columns

**Figure 5. 3 Radars, and Direction data table**

### 5.2.1 Data Preprocessing:

The radar readings are represented as integer values ranging from 0 to 255. These values measure the distance between the car and the object in its path. A higher value means a greater distance between the car and the object, while a lower value represents a smaller distance. The direction column has three classes Straight, Left and Right. The value in this column represents the direction in which the car turned based on the radar readings from the different angles.

Before training a classification model, the data in the table is preprocessed. This involves dropping duplication, removing Null values, and converting the categorical data in the Direction column to numerical data. This can be achieved using an encoder such as a "*to_categorical*" to encode the three classes into a binary numerical representation.

Then, the data is split into input and output variables. The inputs are five radar variable readings from different angles (radar_-60, radar_-30, radar_-0, radar_30, radar_60). The output variable will be the Direction classes.

### 5.2.2 Classification model implementation, and model evaluation:

The model implemented in this work is relatively simple, and not complex, as the classification problem does not have many variables. The model architecture shown in Figure 5. which is visualized in the Python code consists of three hidden layers with four, eight, and eight neurons each, as well as an output layer with two neurons. There are 154 parameters in the model, all of which are trainable.

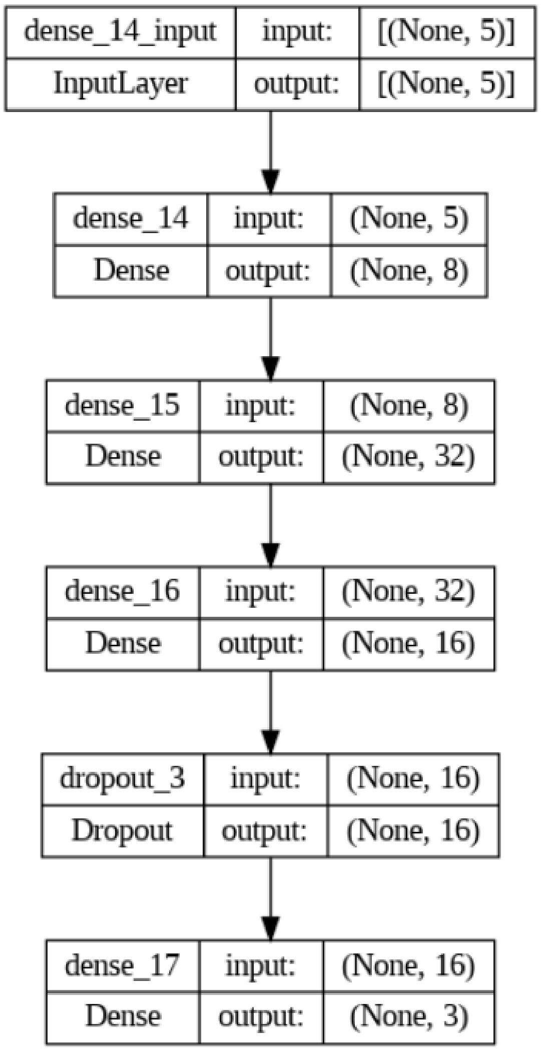
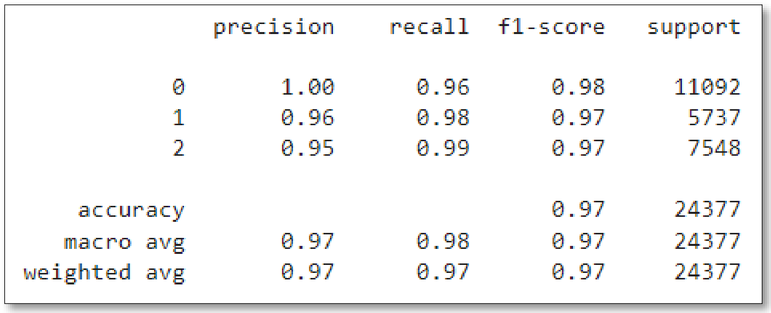

dense_14_input | input: | [(None, 5)]
InputLayer | output: | [(None, 5)]
dense_14 | input: | (None, 5)
Dense | output: | (None, 8)
dense_15 | input: | (None, 8)
Dense | output: | (None, 32)
dense_16 | input: | (None, 32)
Dense | output: | (None, 16)
dropout_3 | input: | (None, 16)
Dropout | output: | (None, 16)
dense_17 | input: | (None, 16)
Dense | output: | (None, 3)

**Figure 5. 4 Neural Network architecture**

After building the neural network, the model is trained using 50 epochs. Loss, precision, recall, and f2_score are used for validation metrics. Those metrics are commonly used in classification model evaluation.

During the model training, it is noticed that the loss metric is decreasing. This indicates that the model learning and the predictions are being improved. On the other hand, precision, recall, and f1_score metrics are improving after each epoch, this indicates that the model accuracy is improving.

As shown in Figure 5.  the model shows an over-accuracy score of 97%. Also, the f1_scroe is 99% which indicates a good overall performance. The Precision is 95% which indicates that 99% of all instances are positive. Finally, the confusion matrix shows that the total predicted instances are 8812 instances.



```
              precision    recall  f1-score   support

           0       1.00      0.96      0.98     11092
           1       0.96      0.98      0.97      5737
           2       0.95      0.99      0.97      7548

    accuracy                           0.97     24377
   macro avg       0.97      0.98      0.97     24377
weighted avg       0.97      0.97      0.97     24377
```

**Figure 5. 5 Model metrics and accuracy**

As shown in Figure 5.  it shows that the training and validation accuracy lines both increase over the set of training. This proves that the model is learning and improving its ability to predict the correct outputs. The validation accuracy reaches a peak of about 95% after about 15 epochs. This suggests that the model is performing well on the validation dataset and is not overfitting to the training data.

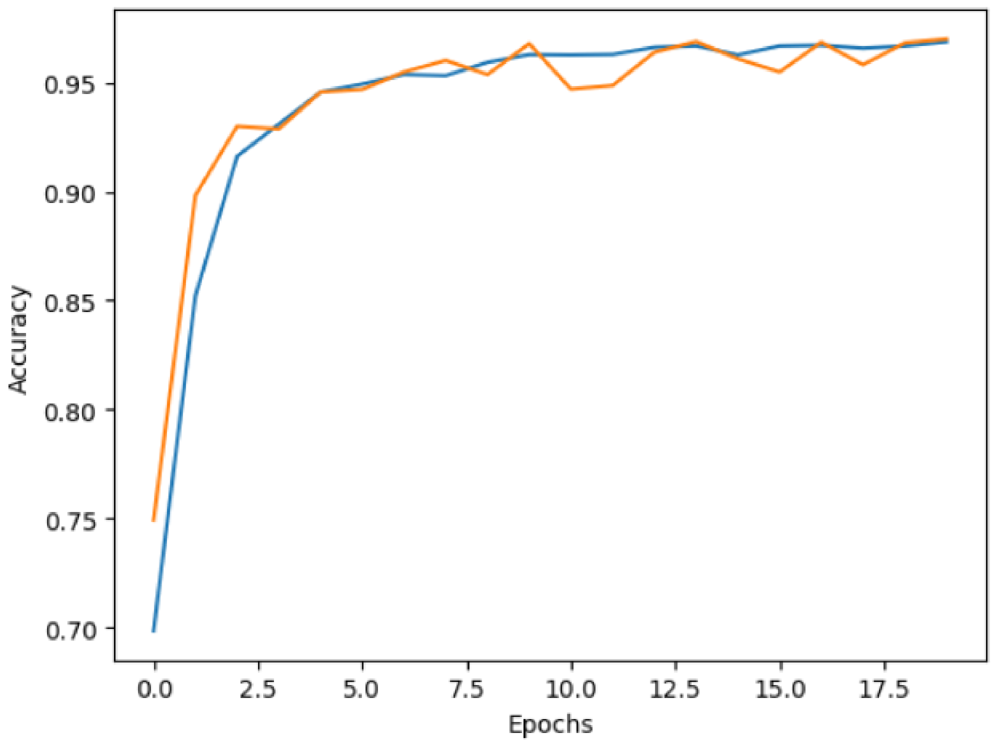


**Figure 5. 6 Graph shows the model's Accuracy vs. Epochs**

The graph shown in Figure 5.  that after about 10 epochs, the training accuracy reaches a plateau of around 0.95. The validation accuracy also reaches a plateau but at a slightly lower level of around 0.92. This suggests that the model has learned the patterns in the training data very well, and is likely to generalize well to new data.

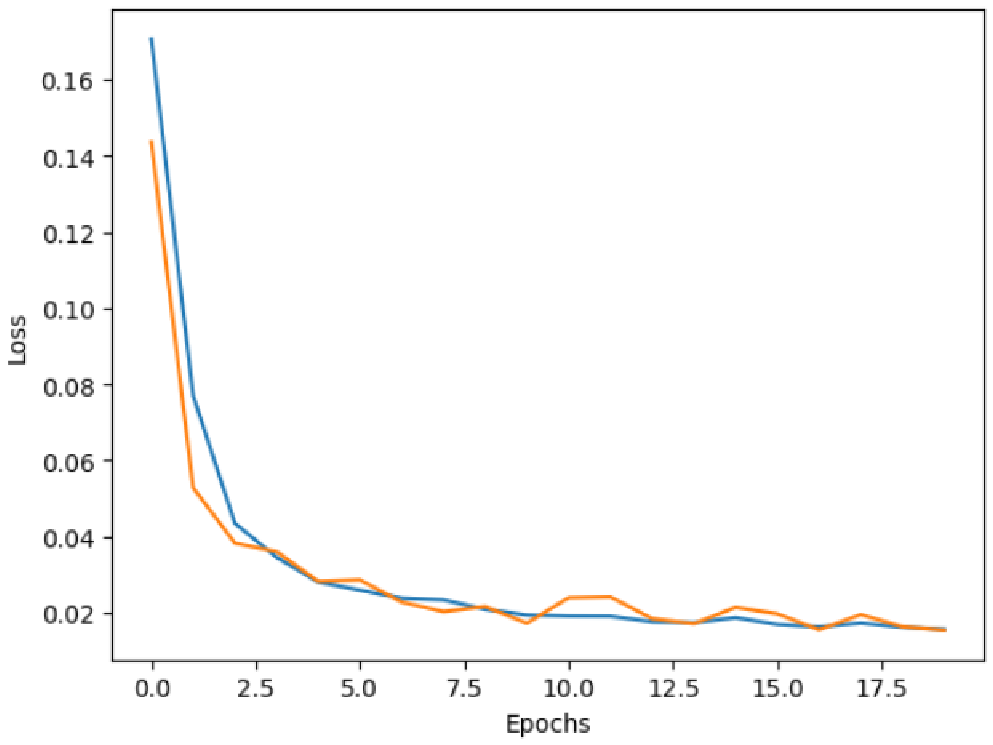
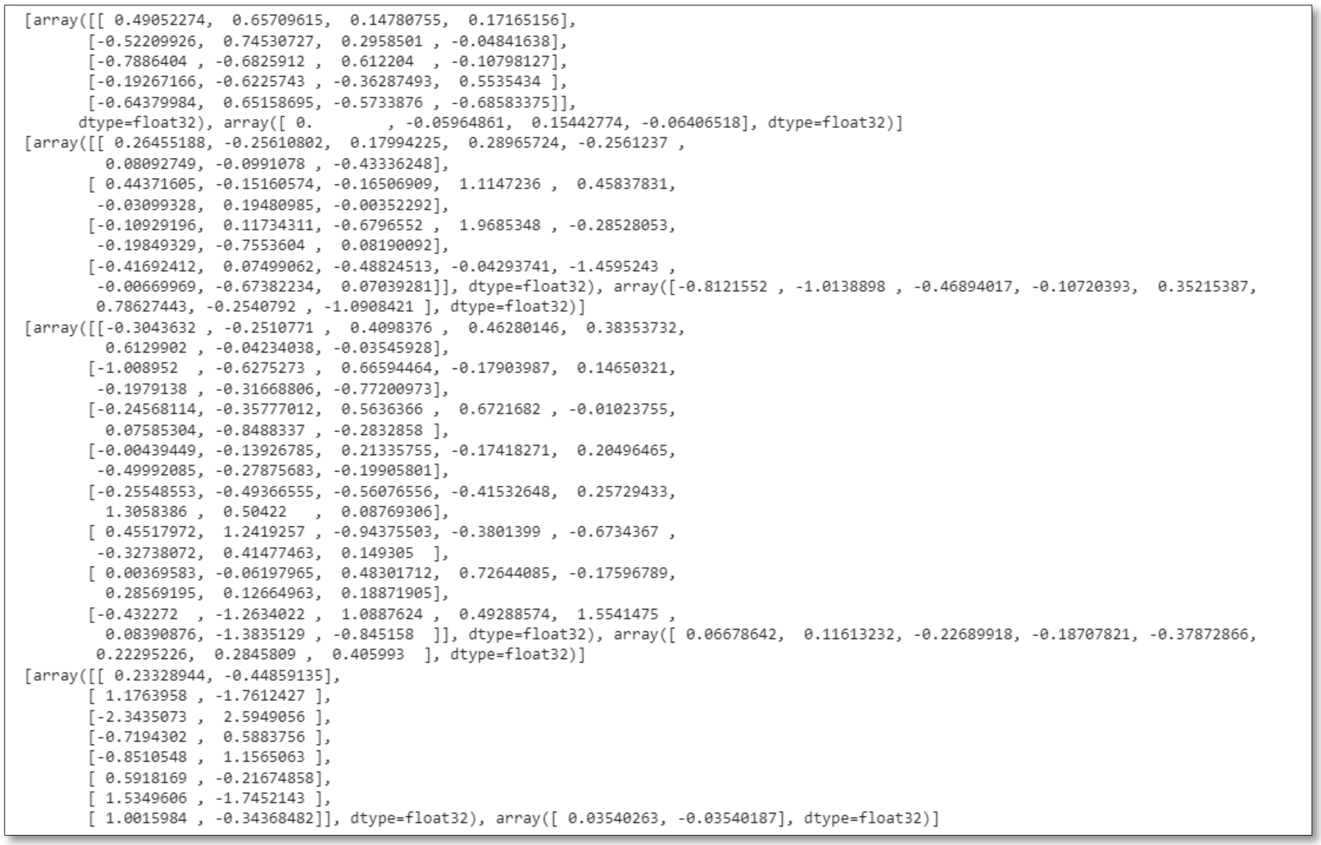


**Figure 5. 7 Graph shows the model's Loss vs. Epochs**

After training the mode the weight and bias can be observed as shown in Figure 5. . Each layer's weight and biases can be accessed with the "*get_weight(*" function, returning a list which consists of two arrays: the first array contains the layer's weight, and the second array contains the layer's biases.



```
[array([[ 0.49052274,  0.65709615,  0.14780755,  0.17165156],
       [-0.52209926,  0.74530727,  0.2958501 , -0.04841638],
       [-0.7886404 , -0.6825912 ,  0.612204  , -0.10798127],
       [-0.19267166, -0.6225743 , -0.36287493,  0.5535434 ],
       [-0.64379984,  0.65158695, -0.5733876 , -0.68583375]],
      dtype=float32), array([ 0.        , -0.05964861,  0.15442774, -0.06406518], dtype=float32)]
[array([[ 0.26455188, -0.25610802,  0.17994225,  0.28965724, -0.2561237 ,
         0.08092749, -0.0991078 , -0.43336248],
       [ 0.44371605, -0.15160574, -0.16506909,  1.1147236 ,  0.45837831,
        -0.03099328,  0.19480985, -0.00352292],
       [-0.10929196,  0.11734311, -0.6796552 ,  1.9685348 , -0.28528053,
        -0.19849329, -0.7553604 ,  0.08190092],
       [-0.41692412,  0.07499062, -0.48824513, -0.04293741, -1.4595243 ,
        -0.00669969, -0.67382234,  0.07039281]], dtype=float32), array([-0.8121552 , -1.0138898 , -0.46894017, -0.10720393,  0.35215387,
        0.78627443, -0.2540792 , -1.0908421 ], dtype=float32)]
[array([[-0.3043632 , -0.2510771 ,  0.4098376 ,  0.46280146,  0.38353732,
         0.6129902 , -0.04234038, -0.03545928],
       [-1.008952  , -0.6275273 ,  0.66594464, -0.17903987,  0.14650321,
        -0.1979138 , -0.31668806, -0.77200973],
       [-0.24568114, -0.35777012,  0.5636366 ,  0.6721682 , -0.01023755,
         0.07585304, -0.8488337 , -0.2832858 ],
       [-0.00439449, -0.13926785,  0.21335755, -0.17418271,  0.20496465,
        -0.49992085, -0.27875683, -0.19905801],
       [-0.25548553, -0.49366555, -0.56076556, -0.41532648,  0.25729433,
         1.3058386 ,  0.50422   ,  0.08769306],
       [ 0.45517972,  1.2419257 , -0.94375503, -0.3801399 , -0.6734367 ,
        -0.32738072,  0.41477463,  0.149305  ],
       [ 0.00369583, -0.06197965,  0.48301712,  0.72644085, -0.17596789,
         0.28569195,  0.12664963,  0.18871905],
       [-0.432272  , -1.2634022 ,  1.0887624 ,  0.49288574,  1.5541475 ,
         0.08390876, -1.3835129 , -0.845158  ]], dtype=float32), array([ 0.06678642,  0.11613232, -0.22689918, -0.18707821, -0.37872866,
        0.22295226,  0.2845809 ,  0.405993  ], dtype=float32)]
[array([[ 0.23328944, -0.44859135],
       [ 1.1763958 , -1.7612427 ],
       [-2.3435073 ,  2.5949056 ],
       [-0.7194302 ,  0.5883756 ],
       [-0.8510548 ,  1.1565063 ],
       [ 0.5918169 , -0.21674858],
       [ 1.5349606 , -1.7452143 ],
       [ 1.0015984 , -0.34368482]], dtype=float32), array([ 0.03540263, -0.03540187], dtype=float32)]
```

**Figure 5. 8 Model weight and biases**

### 5.2.3 TFLite model saving:

The final step in creating the model that shall be deployed on the STM32F429I-DISCOVERY is converting the built Keras model to the TensorFlow Lite model.

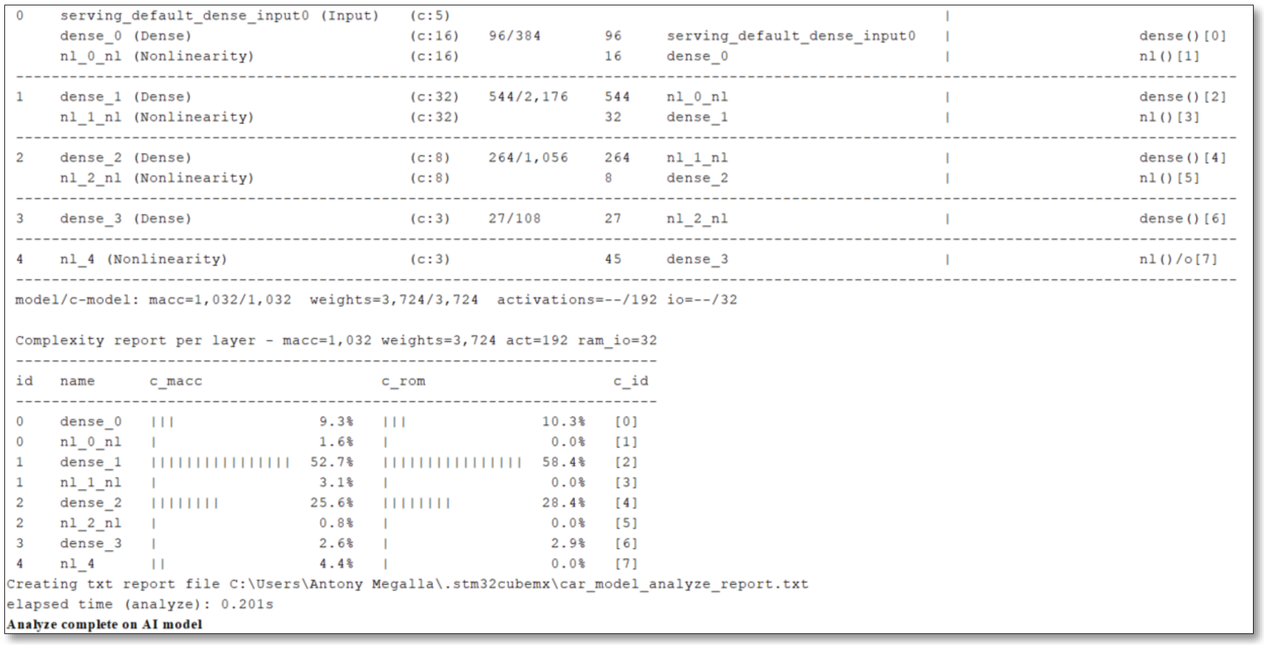
When the TFLite model is successfully converted, the accuracy is measured, and the result is 97% accuracy. Then, it was tested by running inferences and giving the same predicted output as the Keras model. Which also is concluded as a successful conversion without losing much accuracy.

After converting the model to the TFLite model it is observed the model's physical file weight is decreased from 96 Kbytes to 16 Kbytes, which means that the model is successfully reduced and quantized.

## 5.3 Deploying AI model on STM32F429I-DISCOVERY

STM32F429I-DISCOVERY X-cube-AI tool is used in analyzing the model, which gives some information about relative complexity and how much RAM is needed, as shown in Figure

5.9. The estimated memory usage depends on the complexity of the model, the data type used for weights, and the target hardware's memory constraints.



```
0    serving_default_dense_input0 (Input)    (c:5)                                                    |
     dense_0 (Dense)                         (c:16)   96/384       96     serving_default_dense_input0  |                    dense()[0]
     nl_0_nl (Nonlinearity)                  (c:16)                16     dense_0                       |                    nl()[1]
------------------------------------------------------------------------------------------------------------------------------------
1    dense_1 (Dense)                         (c:32)   544/2,176    544    nl_0_nl                       |                    dense()[2]
     nl_1_nl (Nonlinearity)                  (c:32)                32     dense_1                       |                    nl()[3]
------------------------------------------------------------------------------------------------------------------------------------
2    dense_2 (Dense)                         (c:8)    264/1,056    264    nl_1_nl                       |                    dense()[4]
     nl_2_nl (Nonlinearity)                  (c:8)                 8      dense_2                       |                    nl()[5]
------------------------------------------------------------------------------------------------------------------------------------
3    dense_3 (Dense)                         (c:3)    27/108       27     nl_2_nl                       |                    dense()[6]
------------------------------------------------------------------------------------------------------------------------------------
4    nl_4 (Nonlinearity)                     (c:3)                 45     dense_3                       |                    nl()/o[7]
------------------------------------------------------------------------------------------------------------------------------------
model/c-model: macc=1,032/1,032  weights=3,724/3,724  activations=--/192 io=--/32

Complexity report per layer - macc=1,032 weights=3,724 act=192 ram_io=32
------------------------------------------------------------------------
id   name      c_macc                     c_rom                     c_id
------------------------------------------------------------------------
0    dense_0   |||                 9.3%   |||                10.3%   [0]
0    nl_0_nl   |                   1.6%   |                   0.0%   [1]
1    dense_1   ||||||||||||||||   52.7%   ||||||||||||||||   58.4%   [2]
1    nl_1_nl   |                   3.1%   |                   0.0%   [3]
2    dense_2   ||||||||           25.6%   ||||||||           28.4%   [4]
2    nl_2_nl   |                   0.8%   |                   0.0%   [5]
3    dense_3   |                   2.6%   |                   2.9%   [6]
4    nl_4      ||                  4.4%   |                   0.0%   [7]
Creating txt report file C:\Users\Antony Megalla\.stm32cubemx\car_model_analyze_report.txt
elapsed time (analyze): 0.201s
Analyze complete on AI model
```

**Figure 5. 9 Model analysis using STM32 visualization tools**

## 5.4 Testing the system on Different GUI environments

The model was successfully deployed to provide self-driving capabilities for the car on the track GUI after training and testing it on the same track, as shown in Figure 5. . Using pixel colors, the car's five radars determined whether it was on track. The track was grey, with green grass margins. The model was successful in enabling the car to drive itself.

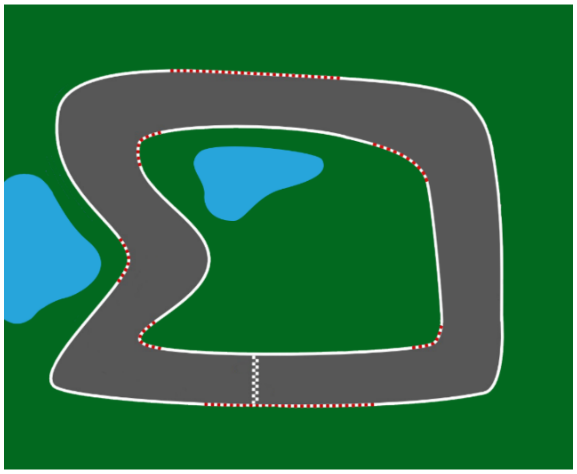


**Figure 5. 10 GUI car track used in training and testing**

Furthermore, the deployed model was evaluated on tracks other than the training model track, as shown in Figure 5.. Despite the change in the image of the track, the neural network successfully enabled the car to self-drive on the new tracks.

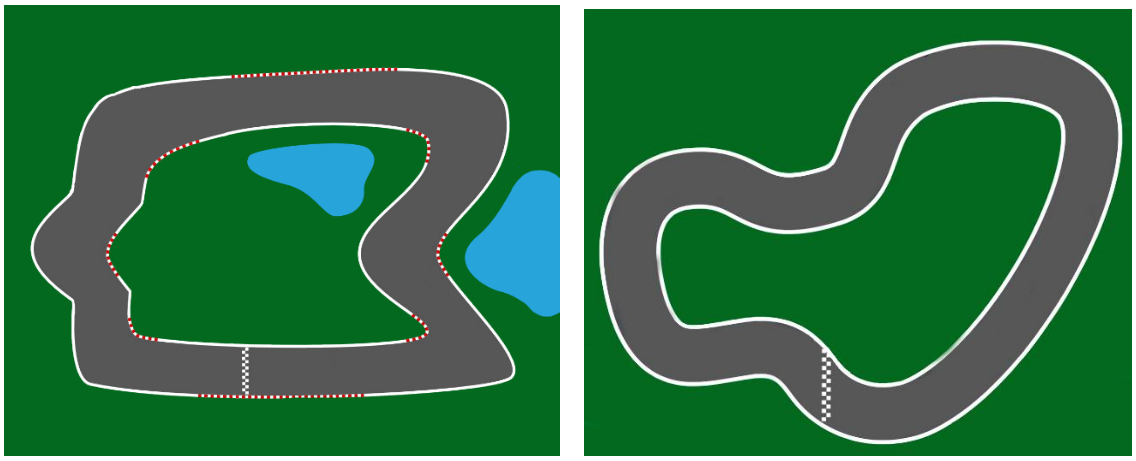


**Figure 5.11 GUI different car tracks**

These results demonstrated the model's adaptability to different contexts and handling of shifting situations.

# Chapter 6: Conclusion and Future Work

## 6.1 Conclusion

This thesis delves into the world of artificial neural networks (ANN) and their integration into low-power microcontrollers, with a specific focus on a computerized self-driven car simulator.

Chapter 1, introduces related works, and fundamental ANN concepts, and outlines the thesis objectives. Chapter 2 gets hands-on, creating a Python-based car simulator using the PyGame library for visually engaging car graphics. The neural network, coded in Python with Keras TensorFlow, brings the self-driving car to life in a racetrack setting.

Chapter 3 takes a detour into the hardware realm, exploring STM32F429I-DISCOVERY. Here, not only discuss physical connections but also delve into the study of ADC and DAC possibilities. This section provides crucial insights into testing circuits and understanding how artificial neuron hardware behaves in real-life scenarios.

Switching back to the software in Chapter 4, navigates through implementing the graphical user interface (G.U.I.) car simulator. This involves establishing communication channels between the G.U.I. and microcontroller using serial communication and UART protocol. The chapter concludes with successful deploying the machine learning model on the STM32F429I-DISCOVERY, showcasing effective communication with the GUI.

Chapters 5 and 6 are the results, conclusions, and future considerations. Through figures and tables. These results highlight the potential of deploying artificial neural networks on low-power microcontrollers, pushing the boundaries of machine learning applications. However, challenges persist, emphasizing the ongoing need to optimize algorithms and address safety concerns.

Overall, this thesis work provides valuable insights into the development of artificial neuron systems on low-power microcontrollers for self-driving cars and lays a foundation for future research in this field.

## 6.2 Future work

Looking ahead to future work recommendations include the following:

- Exploring more advanced communication protocols between software simulators GUI and hardware devices under test STM32 microcontrollers.
- Optimizing neural network algorithms for improved performance.
- Testing the system as a test environment for the ongoing Artificial Neuron (AN) research projects in the Department of Electron Devices in BME.
- Testing these systems in real-world driving scenarios.

# Acknowledgement

This thesis work owes its successful completion to the invaluable support and guidance of Dr. Peter Neumann, my dedicated supervisor. His patience, encouragement, and insightful advice were instrumental throughout the entire year-long journey. I feel fortunate to have had a supervisor who not only cared deeply about my work but also promptly addressed my queries and concerns.

I extend my sincere thanks to the University of Budapest for Technology and Economics for providing the platform for my Master's degree studies. This opportunity not only allowed me to pursue academic excellence but also served as a catalyst for the development of my career and the enhancement of my technical knowledge.

I would also like to acknowledge Dr. Neumann for not only assisting with my thesis but also for his encouragement and support in my decision to apply for the TDK challenge. His belief in my capabilities has been a motivating factor in taking on new challenges and pushing the boundaries of my academic and professional endeavors.

# References

[1] Z. Shao, X. Cao, H. Luo, and P. Jin, "Recent progress in the phase-transition mechanism and modulation of vanadium dioxide materials," *NPG Asia Materials 2018 10:7*, vol. 10, no. 7, pp. 581–605, Jul. 2018, doi: 10.1038/s41427-018-0061-2.

[2] J. Mizsei, M. C. Bein, J. Lappalainen, L. Juhász, and B. Plesz, "The Phonsistor – A Novel VO2 Based Nanoscale Thermal-electronic Device and Its Application in Thermal-electronic Logic Circuits (TELC)," *Mater Today Proc*, vol. 2, no. 8, pp. 4272–4279, Jan. 2015, doi: 10.1016/J.MATPR.2015.09.013.

[3] W. Yi, K. K. Tsang, S. K. Lam, X. Bai, J. A. Crowell, and E. A. Flores, "Biological plausibility and stochasticity in scalable VO2 active memristor neurons," *Nat Commun*, vol. 9, no. 1, Sep. 2018, doi: 10.1038/s41467-018-07052-w.

[4] Q. Zhang, H. Yu, M. Barbiero, B. Wang, and M. Gu, "Artificial neural networks enabled by nanophotonics," *Light: Science & Applications 2019 8:1*, vol. 8, no. 1, pp. 1–14, May 2019, doi: 10.1038/s41377-019-0151-0.

[5] D. Kuan, G. Phipps, and C. Hsueh, "Autonomous Robotic Vehicle Road Following," *IEEE Trans Pattern Anal Mach Intell*, vol. 10, no. 5, pp. 648–658, 1988, doi: 10.1109/34.6773.

[6] STM32 Official Website, "STM32 Microcontrollers (MCUs) - STMicroelectronics." Accessed: Sep. 29, 2023. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html

[7] J. Lappalainen, J. Mizsei, and M. Huotari, "Neuromorphic thermal-electric circuits based on phase-change VO 2 thin-film memristor elements," *J Appl Phys*, vol. 125, no. 4, Jan. 2019, doi: 10.1063/1.5037990/156327.

[8] J. Mizsei and J. Lappalainen, "Microelectronics, Nanoelectronics: Step behind the red brick wall using the thermal domain," *Mater Today Proc*, vol. 7, pp. 888–893, Jan. 2018, doi: 10.1016/j.matpr.2018.12.089.

[9] R. R. Navthar and I. Dr. Vithalrao Vikhe Patil College of Engineering, "(PDF) Analysis of Oil Film Thickness in Hydrodynamic Journal Bearing Using Artificial Neural Networks." Accessed: Nov. 25, 2022. [Online]. Available: https://www.researchgate.net/publication/220027404_Analysis_of_Oil_Film_Thickness_in_Hydrodynamic_Journal_Bearing_Using_Artificial_Neural_Networks

[10] V. Gs, "(PDF) Artificial Neural Network based Condition Monitoring of Rolling Element Bearing." Accessed: Nov. 25, 2022. [Online]. Available: https://www.researchgate.net/publication/263011787_Artificial_Neural_Network_based_Condition_Monitoring_of_Rolling_Element_Bearing

[11] J. I. BM Wilamowski, "Intelligent Systems - Google Books." Accessed: Nov. 25, 2022. [Online]. Available: https://books.google.hu/books?hl=en&lr=&id=fn50DwAAQBAJ&oi=fnd&pg=PT10&dq=Neural+networks:+A+requirement+for+intelligent+systems&ots=ZEE2sM35ya&sig=4nKFNXUw10HTXvo1Hi7FSB5Dqsk&redir_esc=y#v=onepage&q=Neural networks%3A A requirement for intelligent system

[12] P. Jula, "Applications of Artificial Neural Networks in Interactive Simulation," *Masters Theses*, Aug. 1996, Accessed: Nov. 23, 2022. [Online]. Available: https://scholarworks.wmich.edu/masters_theses/5046

[13] "Finding the Cost Function of Neural Networks | by Chi-Feng Wang | Towards Data Science." Accessed: May 21, 2022. [Online]. Available: https://towardsdatascience.com/step-by-step-the-math-behind-neural-networks-490dc1f3cfd9

[14] A. Zell, N. Mache, T. Sommer, and T. Korb, “Recent developments of the SNNS neural network simulator,” *SPIE*, vol. 1469, pp. 708–718, Aug. 1991, doi: 10.1117/12.45009.

[15] P. W. & D. Situnayake, “TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power ... - Pete Warden, Daniel Situnayake - Google Books.” Accessed: Nov. 23, 2022. [Online]. Available: https://books.google.hu/books?hl=en&lr=&id=tn3EDwAAQBAJ&oi=fnd&pg=PP1&dq=Why+Machine+Learning+on+Microcontrollers%3F&ots=jprq9t35z6&sig=d1alXkO8oC9HjWPLlIGo8Uif-cc&redir_esc=y#v=onepage&q=Why Machine Learning on Microcontrollers%3F&f=false

[16] Y. Yusof, H. M. A. H. Mansor, and H. M. D. Baba, “Simulation of mobile robot navigation utilizing reinforcement and unsupervised weightless neural network learning algorithm,” *2015 IEEE Student Conference on Research and Development, SCOReD 2015*, pp. 123–128, 2015, doi: 10.1109/SCORED.2015.7449308.

[17] S. Ledesma, M. A. Ibarra-Manzano, M. G. Garcia-Hernandez, and D. L. Almanza-Ojeda, “Neural lab a simulator for artificial neural networks,” *Proceedings of Computing Conference 2017*, vol. 2018-January, pp. 716–721, Jan. 2018, doi: 10.1109/SAI.2017.8252175.

[18] J.-S. Park, H.-J. Lee, D.-Y. Hwang, and S. Cho, “Pacman Game Reinforcement Learning Using Artificial Neural-network and Genetic Algorithm,” *IEMEK Journal of Embedded Systems and Applications*, vol. 15, no. 5, pp. 261–268, 2020, doi: 10.14372/IEMEK.2020.15.5.261.

[19] E. Heiden, D. Millard, E. Coumans, Y. Sheng, and G. S. Sukhatme, “NeuralSim: Augmenting Differentiable Simulators with Neural Networks,” *Proc IEEE Int Conf Robot Autom*, vol. 2021-May, pp. 9474–9481, Nov. 2020, doi: 10.48550/arxiv.2011.04217.

[20] B. V. Uba, “Smart robot control information technology,” 2021, Accessed: Nov. 26, 2022. [Online]. Available: https://essuir.sumdu.edu.ua/handle/123456789/86925

[21] A. Özeloğlu, İ. G. Gürbüz, and İ. San, “Deep reinforcement learning-based autonomous parking design with neural network compute accelerators,” *Concurr Comput*, vol. 34, no. 9, p. e6670, Apr. 2022, doi: 10.1002/CPE.6670.

[22] G. Brockman *et al.*, “OpenAI Gym,” Jun. 2016, Accessed: Oct. 15, 2023. [Online]. Available: https://arxiv.org/abs/1606.01540v1

[23] H. Hazan *et al.*, “BindsNET: A machine learning-oriented spiking neural networks library in python,” *Front Neuroinform*, vol. 12, p. 409297, Dec. 2018, doi: 10.3389/FNINF.2018.00089/BIBTEX.

[24] P. Jula, A. Houshyar, … F. S.-C. & industrial, and undefined 1996, “Application of artificial neural networks in interactive simulation,” *Elsevier*, 1996, Accessed: Nov. 25, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0360835296001659

[25] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, “CARLA: An Open Urban Driving Simulator.” PMLR, pp. 1–16, Oct. 18, 2017. Accessed: Nov. 25, 2022. [Online]. Available: https://proceedings.mlr.press/v78/dosovitskiy17a.html

[26] R. J.-O. D. Conference and undefined 2005, “Rapid game development in Python,” *remwebdevelopment.com*, Accessed: Nov. 26, 2022. [Online]. Available: https://www.remwebdevelopment.com/uploadedFiles/file_1409147263.pdf

[27] “32F429IDISCOVERY - Discovery kit with STM32F429ZI MCU * New order code STM32F429I-DISC1 (replaces STM32F429I-DISCO) - STMicroelectronics.” Accessed: Apr. 30, 2023. [Online]. Available: https://www.st.com/en/evaluation-tools/32f429idiscovery.html#overview

[28] Rachel Hinrichs, “A circular buffer allows asynchronous write and read operations and can... | Download Scientific Diagram.” Accessed: Nov. 24, 2022. [Online]. Available: https://www.researchgate.net/figure/A-circular-buffer-allows-asynchronous-write-and-read-operations-and-can-store-N-samples_fig6_334288006

[29] H. Abbas, I. Abbassi, … A. A.-I. T., and undefined 2022, “2022 Index IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems Vol. 41,” *ieeexplore.ieee.org*, Accessed: Oct. 16, 2023. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9982293/

[30] S. Zhang, A. C.-I. T. on C. and, and undefined 1992, “Lagrange programming neural networks,” *researchgate.net*, vol. 11, no. 7, 1992, doi: 10.1109/82.160169.

[31] [Online], “Get started | PyCharm.” Accessed: Nov. 27, 2022. [Online]. Available: https://www.jetbrains.com/help/pycharm/quick-start-guide.html

[32] [Online], “CodeReclaimers/neat-python: Python implementation of the NEAT neuroevolution algorithm.” Accessed: Dec. 06, 2022. [Online]. Available: https://github.com/CodeReclaimers/neat-python

[33] C. Bailey, “Adaptability of Improved NEAT in Variable Environments,” Oct. 2021, doi: 10.48550/arxiv.2201.07977.

## Appendix A: Create a game, display car, and driving functions, radars, and collision points and adjust user inputs.

```
'''
The Purpose of the radar is to measure the distance between the car and the edge of
the track
This data will be used as an input to the network
'''
import pygame
import os
import math
import sys
import csv

def drive(self):
    if self.drive_state:
        self.rect.center += self.vel_vector * 6

def collision(self):
    length = 40
    collision_point_right = [int(self.rect.center[0] +
math.cos(math.radians(self.angle + 18)) * length),
    int(self.rect.center[1] - math.sin(math.radians(self.angle + 18)) * length)]
    collision_point_left = [int(self.rect.center[0] +
math.cos(math.radians(self.angle - 18)) * length),
    int(self.rect.center[1] - math.sin(math.radians(self.angle - 18)) * length)]

def radar(self, radar_angle):
    length = 0
    x = int(self.rect.center[0])
    y = int(self.rect.center[1])

    while (0 <= x < SCREEN.get_width()) and (0 <= y < SCREEN.get_height()) and not
SCREEN.get_at(
        (x, y)) == pygame.Color(2, 105, 31, 255) and length < 200:
        length += 1
        x = int(self.rect.center[0] + math.cos(math.radians(self.angle +
radar_angle)) * length)
        y = int(self.rect.center[1] - math.sin(math.radians(self.angle +
radar_angle)) * length)

def eval_genomes():
    run = True
    while run:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()
        car.draw(SCREEN)
        car.update()
        pygame.display.update()

eval_genomes()
```

## Appendix B: STM32F429I-DISCOVERY C code implementation using STM32CubeIDE

```
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow import keras
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, MinMaxScaler
from sklearn.metrics import classification_report, confusion_matrix

# Load and preprocess the dataset
dataset = pd.read_csv("Data.csv")
X = dataset.iloc[:, 0:5].values
y = dataset.iloc[:, 5].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=0)
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create and train a neural network model
model = keras.Sequential([
    keras.layers.Input(shape=(5,)),
    keras.layers.Dense(16, activation='relu'),
    keras.layers.Dense(32, activation='relu'),
    keras.layers.Dropout(0.2),
    keras.layers.Dense(8, activation='relu'),
    keras.layers.Dense(3, activation='softmax')
])
model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.001), loss='mse',
metrics=['accuracy'])
history = model.fit(X_train, keras.utils.to_categorical(y_train, num_classes=3),
                    epochs=50, batch_size=32, verbose=1, validation_split=0.2)

# Evaluate the model on the test set
loss, accuracy = model.evaluate(X_test, keras.utils.to_categorical(y_test,
num_classes=3))
print(f'Test Accuracy: {accuracy * 100:.2f}%')

# Convert the model to TFLite
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()
open('new_model.tflite', 'wb').write(tflite_model)
```

# Appendix C: STM32F429I-DISCOVERY C code implementation using STM32CubeIDE

```
int main(void)
{
  /* USER CODE BEGIN 1 */
        char buf[50];

        int buf_len = 0;
        ai_error ai_err;
        ai_i32 nbatch;
        uint32_t timestamp;
        float y_val[2];
        uint16_t array[5];
        uint8_t buffer[10];

        // Chunk of memory used to hold intermediate values for neural network
        AI_ALIGNED(4) ai_u8 activations[AI_CAR_MODEL_DATA_ACTIVATIONS_SIZE];

        // Buffers used to store input and output tensors
        AI_ALIGNED(4) ai_i8 in_data[AI_CAR_MODEL_IN_1_SIZE_BYTES*5];
        AI_ALIGNED(4) ai_i8 out_data[AI_CAR_MODEL_OUT_1_SIZE_BYTES*2];

        // Pointer to our model
        ai_handle car_model = AI_HANDLE_NULL;

        // Initialize wrapper structs that hold pointers to data and info about
the
        // data (tensor height, width, channels)
        ai_buffer ai_input[5] = AI_CAR_MODEL_IN;
        ai_buffer ai_output[2] = AI_CAR_MODEL_OUT;

        // Set working memory and get weights/biases from model
        ai_network_params ai_params =
AI_NETWORK_PARAMS_INIT(AI_CAR_MODEL_DATA_WEIGHTS(ai_car_model_data_weights_get()),

        AI_CAR_MODEL_DATA_ACTIVATIONS(activations));

        // Set pointers wrapper structs to our data buffers
         for (int i = 0; i < 5; i++) {
           ai_input[i].n_batches = 1;
           ai_input[i].data = AI_HANDLE_PTR(&in_data[i *
AI_CAR_MODEL_IN_1_SIZE_BYTES]);
         }
         for (int i = 0; i < 2; i++) {
           ai_output[i].n_batches = 1;
           ai_output[i].data = AI_HANDLE_PTR(&out_data[i *
AI_CAR_MODEL_OUT_1_SIZE_BYTES]);
         }

  // Start timer/counter
   HAL_TIM_Base_Start(&htim1);

  while (1)
  {
```

```c
// Wait for START_BIT (0x01)
uint8_t start_byte;
while (1)
{
    HAL_UART_Receive(&huart1, &start_byte, 1, HAL_MAX_DELAY);
    if (start_byte == 0x01)
    {
        break;
    }
}

// Receive data until STOP_BIT (0x02) is received
uint8_t buffer[10];
int index = 0;
while (1)
{
    HAL_UART_Receive(&huart1, &buffer[index], 1, HAL_MAX_DELAY);
    if (buffer[index] == 0x02)
    {
        break;
    }
    index++;
    if (index >= 10)
    {
        // Handle error: STOP_BIT was not received
        break;
    }
}

// Convert the received bytes to integers
int16_t array[5];
float arrayDac[5];
array[0] = (buffer[0] << 8) | buffer[1];
array[1] = (buffer[2] << 8) | buffer[3];
array[2] = (buffer[4] << 8) | buffer[5];
array[3] = (buffer[6] << 8) | buffer[7];
array[4] = (buffer[8] << 8) | buffer[9];

((ai_float *)in_data)[0] = (ai_float)array[0];
((ai_float *)in_data)[1] = (ai_float)array[1];
((ai_float *)in_data)[2] = (ai_float)array[2];
((ai_float *)in_data)[3] = (ai_float)array[3];
((ai_float *)in_data)[4] = (ai_float)array[4];

  // Get current timestamp
  timestamp = htim1.Instance->CNT;

  // Perform inference
  nbatch = ai_car_model_run(car_model, &ai_input[0], &ai_output[0]);
  if (nbatch != 1) {
    buf_len = sprintf(buf, "Error: could not run inference\r\n");
    HAL_UART_Transmit(&huart1, (uint8_t *)buf, buf_len, 100);
    continue; // jump to next iteration if inference fails
  }

  // Read output (predicted y) of neural network
```

```
              float y_vals[3];
              for (uint32_t i = 0; i <= 2; i++)
              {
                y_vals[i] = ((float *)out_data)[i];
              }

            char transmit_data;

            if (y_vals[0] > 0.7)
            {
              if (y_vals[1] < 0.7 && y_vals[2] < 0.7)
              {
                     transmit_data = '0';
              }
            }
            if (y_vals[1] > 0.7)
              {
              if (y_vals[0] < 0.7 && y_vals[2] < 0.7)
              {
                     transmit_data = '1';
              }
              }
            if (y_vals[2] > 0.7)
              {
              if (y_vals[0] < 0.7 && y_vals[1] < 0.7)
                     {
                     transmit_data = '2';
                     }
              }
            // Print the transmit data using HAL transmit
            HAL_UART_Transmit(&huart1, (uint8_t *)&transmit_data, 1, HAL_MAX_DELAY);

    }
  }
```

## Appendix D: Python code modifications to communicate with the deployed model on STM32F429I-DISCOVERY

```
def eval_genomes():
run = True
state = 0
ser_send = serial.Serial(port='COM5', baudrate=115200)

# Add start and end bits
start_bit = b'\x01'
end_bit = b'\x02'

while run:

# Generate a new array of 5 random integers between 0 and 65535
array = [car.sprite.data()[0], car.sprite.data()[1], car.sprite.data()[2],
car.sprite.data()[3],
car.sprite.data()[4]]
# Pack each integer value into 2 bytes in big-endian format
values = [x.to_bytes(2, 'big') for x in array]

# Flatten the list of bytes into a single list
bytes_to_send = [b for v in values for b in v]
print(f"Sent Data: {array}")

# Send the start bit, data, and end bit over UART1
ser_send.write(start_bit)
ser_send.write(bytes_to_send)
ser_send.write(end_bit)

# Wait for a response from the C code
response = ser_send.read(1)
integer_value = int.from_bytes(response, byteorder='big')

# User Input
car.sprite.drive_state = True

if integer_value == 48: # 48 => 0
car.sprite.direction = -1
if integer_value == 49: # 49 => 1
car.sprite.direction = 1
if integer_value == 50: # 50=> 2
car.sprite.drive_state = True
```