

TinySpark

Tiny board, big predictions

JC Siderius

Copyright © 2023 JC Siderius

Table of contents

1. TinySpark	3
2. Get started	4
2.1 Introducing TinySpark	4
2.2 TinySpark platform	5
2.3 TinySpark Development kit	7
2.4 Programming the TinyML Development kit	10
3. Chapter 1	15
3.1 Chapter 1 - Introduction to Machine Learning	15
3.2 The Neuron	16
3.3 Logic gates	18
3.4 Logic gates on the TinySpark development board	20
4. Chapter 2	23
4.1 Chapter 2 - The neuron problem	23
4.2 Network connections	24
4.3 Plant monitoring	25
4.4 Plant monitoring on the TinySpark development board	29
5. Chapter 3	31
5.1 Chapter 3 - Training networks	31
5.2 Training networks	32
5.3 Gesture recognition - data aquisition	36
5.4 Gesture recognition - training the model	39
5.5 Gesture recognition - deploying the model	44
6. Beyond TinySpark	46
6.1 Dive into TinyML	46
7. About	48
7.1 TinySpark	48
7.2 TinySpark licenses	49

1. TinySpark

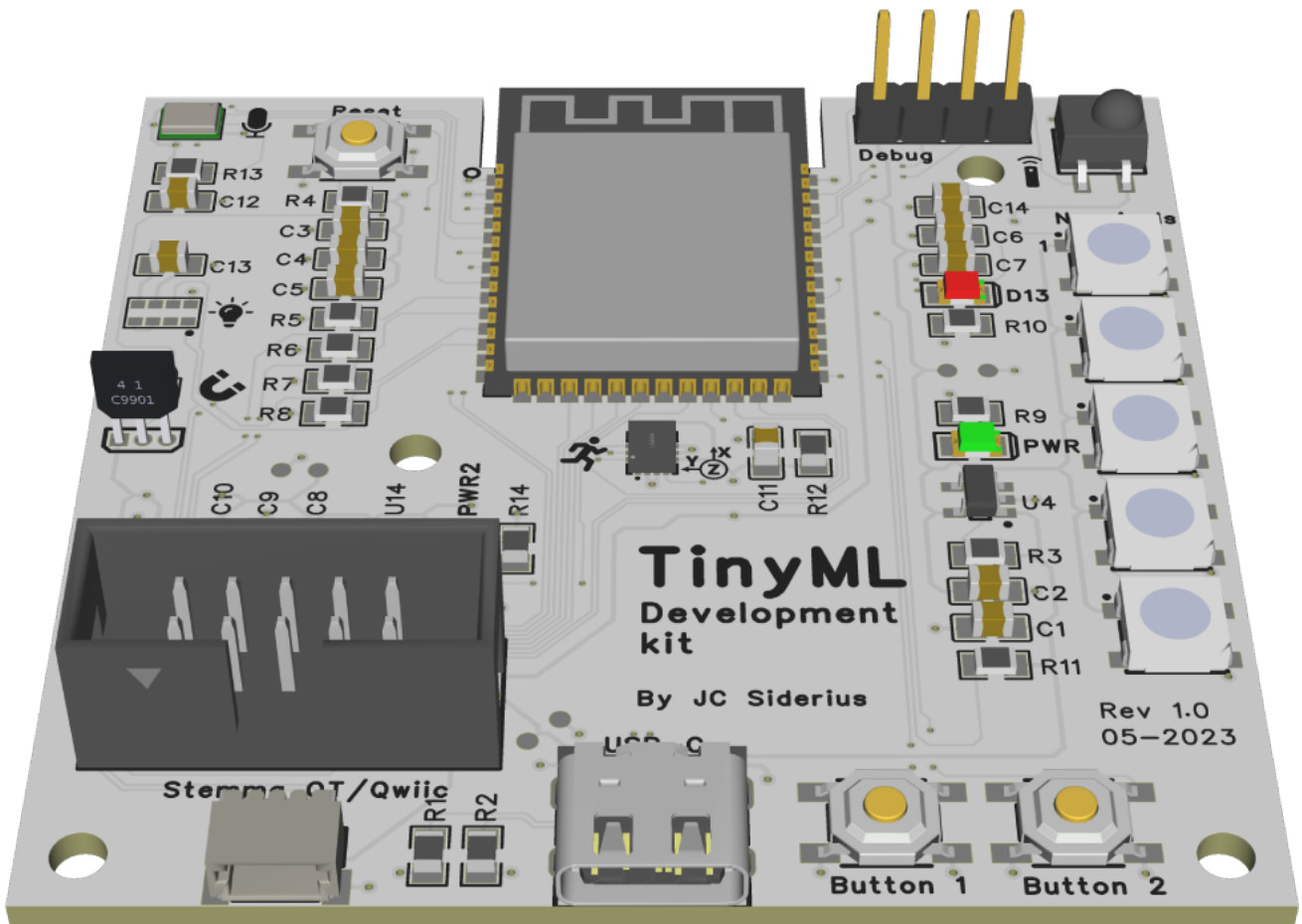
Welcome to **TinySpark**, the online learning platform that teaches you how to create smart machine learning applications on tiny devices.

Tiny Machine Learning (TinyML) is an emerging field of technology that combines deep learning and embedded systems to enable artificial intelligence on resource-constrained hardware. It has many applications in various domains, such as agriculture, environmental sensing, speech recognition, healthcare, security and more.

On the TinySpark platform, you will interactively learn the basics of neural networks, their structure, mathematics and how to train them. The theoretical learning is alternated with mini-projects that deepen your understanding, leading to fun, engaging and interesting insights into TinyML.

To follow along with the projects and examples, you will need a TinySpark development board, which is a tiny device that packs enough processing power and sensors to start your TinyML journey.

[Get started!](#)

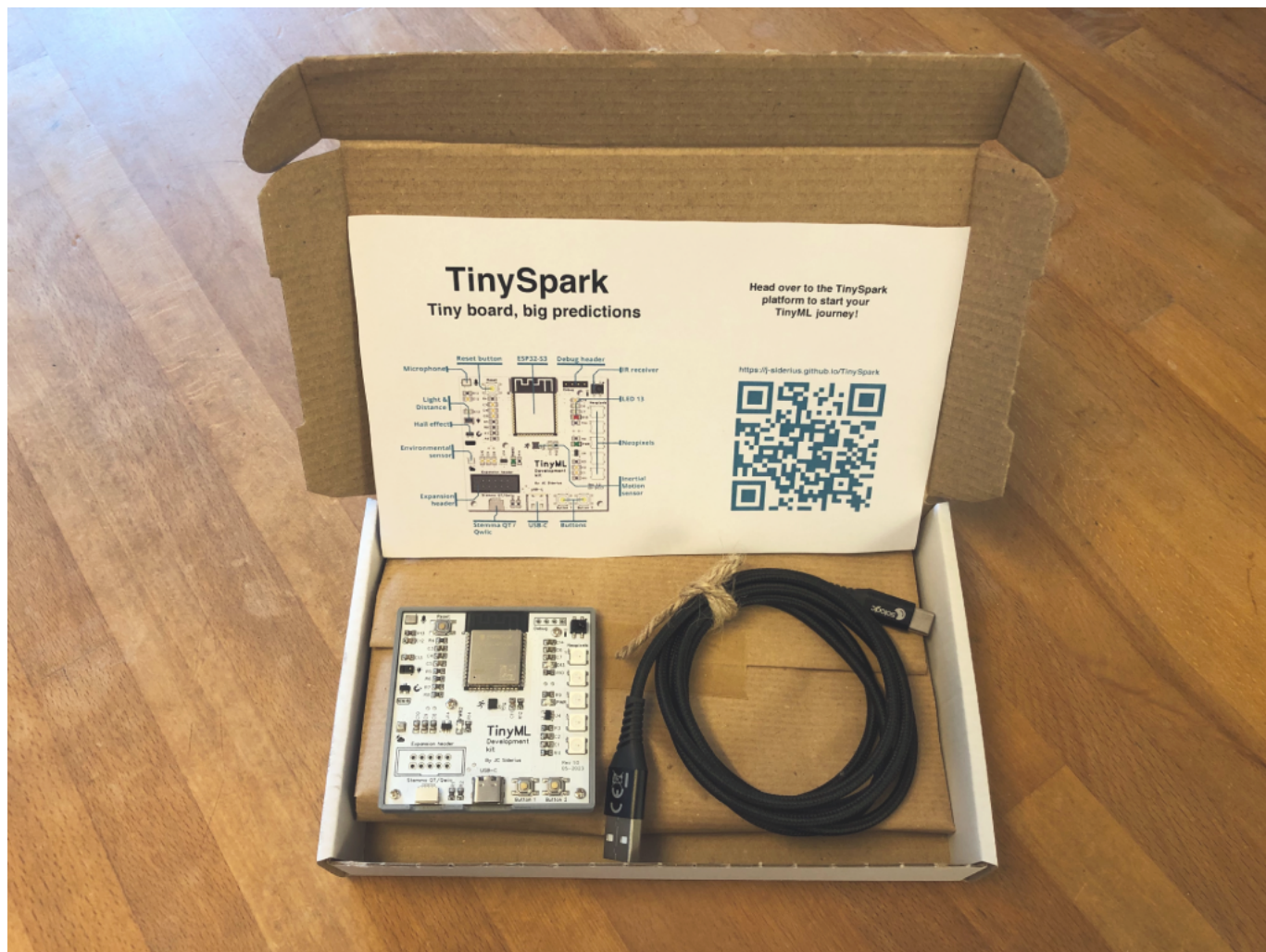


Last updated:

2. Get started

2.1 Introducing TinySpark

These pages give a quick overview of the TinySpark platform and how it works, the TinySpark development kit and all its sensors as well as the programming of neural networks.



In the next section, the TinySpark platform will be explained. Click on [Next](#) to continue the *Get Started* guide.

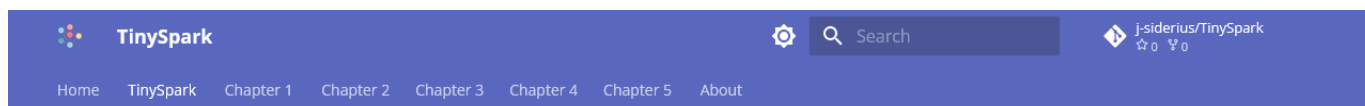
2.2 TinySpark platform

This page gives a quick overview of the the TinySpark platform (this webpage), how it functions, how to use it and where to find help if you need it.

The TinySpark platform is divided into several chapters:

- [Getting started \(this chapter\)](#)
- [Chapter 1: Introduction to neurons](#)
- [Chapter 2: Networks and structures](#)
- [Chapter 3: Training networks](#)
- [Beyond TinySpark](#)

These chapters will introduce various concepts within (Tiny) Machine Learning in an engaging, interactive and project-based way. They can be accessed by clicking on the chapters here, or on the navigation bar at the top of the page (pictured below). Each chapter will explain some theory, let you play around with parts of a neural network and program a mini-project, in which the theory can be put into practise.



To navigate through chapters, it is possible to use the arrows at the bottom of most pages. Clicking the right-arrow will direct you to the next chapter section. The left-arrow can be used to go back to the previous section of a chapter. Additionally, on computers, there is a side-menu available at the top left of each page, which shows all sections of a chapter.



The TinySpark platform uses several methods to teach, for example using textual explanation, formulas, code snippets and interactive applications.

Source code will be displayed on the page, with the option to open Python code in [Google Colaboratory](#), an online code environment for Python notebooks. Any Python code that can be run on a PC (so no TinyML Development Kit code) will be available for testing and playing around on Colab; just click the link and a new notebook will open. If you want to interact with the code, you need a Google account.



test_code.py

```
# This is some Python code
a = 1
b = 2
c = a + b

print(c)
```

TinyML development board code is hosted on Github, since there is no online platform available for running this code. All TinyML code should be uploaded to the development board to see it in action. Further explanation on running code on the TinyML development kit can be found in the [Programming](#) section of this chapter.

**led.py**

```
# Include all libraries
import time
import board
from digitalio import DigitalInOut, Direction

# Initialise LED, declare it an output
led = DigitalInOut(board.LED)
# led = DigitalInOut(board.D13) # Alternatively use the well-known pin 13
led.direction = Direction.OUTPUT

# Every second, flash the LED
while True:
    led.value = 0
    time.sleep(1)
    led.value = 1
    time.sleep(1)
```

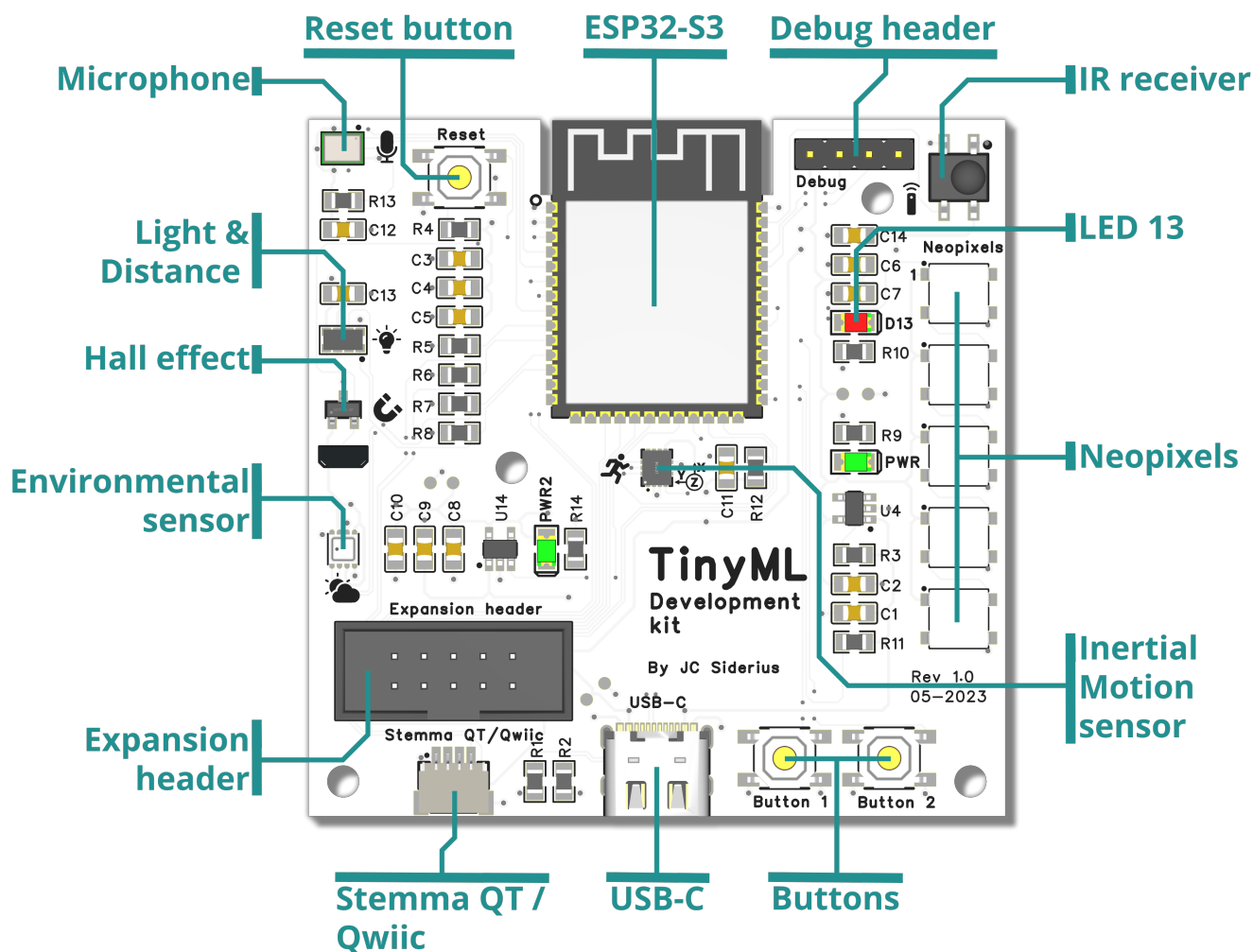
Concepts can also be explained using interactive applications. These allow you to manipulate values, play around with network structures and more, seeing the effects in real time.

Weight 1 0.3

In the next section, the TinyML Development Kit will be introduced. Click on **Next** to continue the *Get Started* guide.

2.3 TinySpark Development kit

The TinySpark Development kit is a development board specifically tailored to the exploration of TinyML concepts and to test possible applications of TinyML in an easy way. It is based on the popular ESP32-S3 chipset, which is perfect for TinyML applications, because of its fast processor speed and large storage. Additionally it has Bluetooth 5 and WiFi capabilities, making it easy to share sensor data and warn users in case of detections or errors. You can learn more about this chip in its [datasheet](#). The full electronic schematic of the TinyML development kit is available [here](#).



The TinyML Development kit contains a selection of sensors which enable direct measurements without needing extra hardware, as well as expansion options to adapt the development kit to your specific projects.

The Development kit contains the following sensors and I/O:

- **Inertial motion sensor:** this sensor measures both the angular motion using a gyroscope, as well as the acceleration using an accelerometer. The datasheet for the LSM6DS3TR-C sensor can be found [here](#). A template code project for using this sensor can be found [here](#).
- **Microphone:** this sensor measures the amplitude and pitch of sound. The datasheet for the ICS-43434 microphone can be found [here](#).
- **Light and Distance sensor:** this sensor measures the ambient light level, as well as short-range distance. The datasheet for the APDS-9930 sensor can be found [here](#). A template code project for using this sensor can be found [here](#).
- **Hall effect sensor:** this sensor measures the magnetic field around it's axis. The datasheet for the AH-49E sensor can be found [here](#). A template code project for using this sensor can be found [here](#).
- **Environmental sensor:** this sensor measures multiple environmental parameters; temperature, relative humidity as well as atmospheric pressure. The datasheet for the BME-280 sensor can be found [here](#). A template code project for using this sensor can be found [here](#).
- **Infrared receiver:** this sensor receives infrared signals, for example from a remote control. The datasheet for the IRM-H638T-TR2 receiver can be found [here](#).
- **Buttons:** two buttons are accessible to the user (Button 1 and Button 2), which can be configured for a multitude of uses. The last button (Reset) is used if the board needs to be debugged, or if you want to restart your board and your application. A template code project for using the buttons can be found [here](#).
- **Output LED:** this Red LED can be used as a simple output, for example to quickly see if your sensor functions correctly. It is connected to the default LED pin, D13. A template code project for using the output LED can be found [here](#).
- **Neopixel LEDs:** the Neopixel LEDs can be used to output application specific information, and since they can be programmed to (individually) display every RGB colour imaginable, you can use them to output loads of useful information. The datasheet for the WS2812B programmable RGB LEDs can be found [here](#). A template code project for using the Neopixels can be found [here](#).
- **USB-C connector:** this connector is used for the main connection to power and program the board. It can be connected using the included USB-C to USB-A cable.
- **Stemma QT / Qwiic connector:** this connector can be used to connect extra sensors to adapt to your specific projects. The connector enables connection to sensors and actuators using the popular [Adafruit Stemma QT](#) or [Sparkfun Qwiic](#) standards. A template code project for using the Stemma QT / Qwiic connector with an external sensor can be found [here](#).
- **Expansion header:** this connector can be used to connect extra peripherals, sensors and actuators to adapt to your specific projects. The connector features power, ground, digital and analog connections, giving you the freedom to connect anything you can think off.

The TinyML development kit also contains some miscellaneous other components such as two green LEDs (for showing power), a Debug header (used to initially program and test the development kit) and some passive components such as resistors and capacitors.

Sensor addresses

Due to the nature of connections to the sensors on the TinySpark development board, the address on which they are reachable might be different depending on the board you have. Each sensor can be configured for a different address in software, to remedy this issue.

The environmental sensor defaults to: `address=0x76`, however it might also be on `address=0x77`.

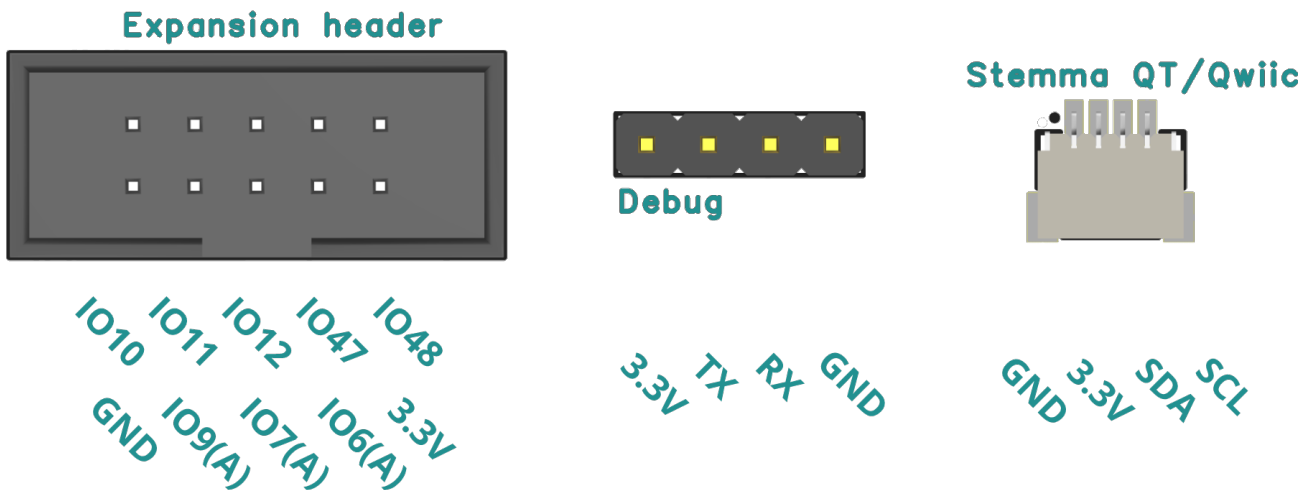
The light and distance sensor defaults to `address=0x39`, this is the only possible address.

The inertial motion sensor defaults to `address=0x6b`, however it might also be on `address=0x6a`.

🔊 Microphone use

Due to driver issues in CircuitPython, it is currently not possible to access the microphone data using this programming language. The microphone is still usable in other programming languages such as [Arduino](#) and the [ESP-IDF](#). Please see the examples in the given links.

There are multiple output pins on the TinyML development board. The specific pinouts are shown in the image below. A template code project for using the expansion pins can be found [here](#).



The Stemma QT / Qwiic connector uses a different I2C communication bus as the main sensors on the board (on board uses I2C₁, Stemma QT / Qwiic uses I2C₂). Please see information below to connect sensors to the Stemma QT / Qwiic connector.

📄 Secondary I2C bus

To connect external sensors to the Stemma QT / Qwiic connector, a new (secondary) I2C bus needs to be initialised. This I2C₂-bus is connected to the following pins: `SDA2=GPI017 SCL2=GPI016`.

To start the secondary I2C bus, use the template project below.

[Github](#) [Open code](#)

secondary_i2c.py

```
# import needed libraries
import busio
from board import *

# define the secondary i2c object with the new pins
i2c2 = busio.I2C(GPI016, GPI017)

# use the i2c2 object below
```

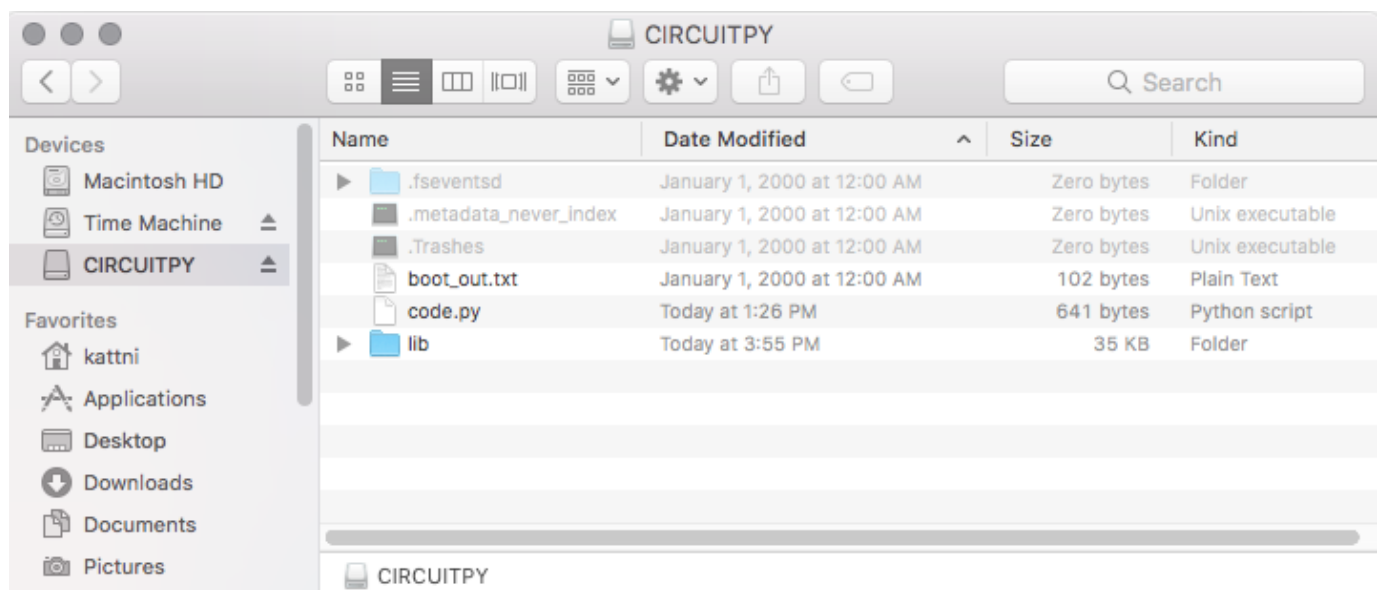
In the next section, programming the TinyML Development kit will be explained.

2.4 Programming the TinyML Development kit

The TinyML Development kit uses the [CircuitPython](#) programming language. It is based on the popular Python 3 programming language, and designed to simplify experimentation and learning on microcontrollers and development kits. It requires very little setup on your computer and since the language is based on Python, code is easy to read and understand. Additionally, there are many libraries available, ready to use in your project.

The TinyML development kit is programmed using the included USB-C to USB-A cable. If your computer requires another connection (e.g. USB-C to USB-C) you have to provide your own cable for connecting the kit.

After connecting the TinyML development kit, two green LEDs on the board should light up, indicating that there is power on the board. At the same time, a new USB-drive named `CIRCUITPY` should appear on your computer (in File Explorer, Finder or Files). This drive contains code, software libraries and files. To learn more about the `CIRCUITPY`-drive, take a look at the [Adafruit - Circuitpy Drive guide](#).



Programming the TinyML development kit is as easy as editing the `code.py` file that is found on the `CIRCUITPY` drive. However, to make coding for the kit easier, the [Code with Mu](#) IDE will be used. Code with Mu works out of the box with CircuitPython, gives helpful programming prompts and was built with learning in mind.

To install the software, head to the [Code with Mu - Downloads](#) page and install the correct version of the IDE for your computer.

Advanced IDE setup

If you are already familiar with programming development kits, or want to work in an IDE which you are familiar with (such as Atom or VS Code), take a look at the [Adafruit - Advanced CircuitPython setup](#).

Be aware that all examples are based on the Code with Mu editor, they should function in other editors as well. Setting up the serial connection to the microcontroller can be somewhat more difficult however.

After installing, open the Code with Mu editor and (upon first start) select the CircuitPython mode. This ensures that the IDE is set up correctly for use with the TinyML development kit.

Reset the Code with Mu editor

If you already started Code with Mu and did not select the CircuitPython mode, click the **Mode** button in the top left of the Code with Mu editor, and select CircuitPython. If you want more information about setting up Code with Mu for CircuitPython, take a look at the [Code with Mu - CircuitPython setup](#).



Now you can start coding. To begin, try to make the built-in LED (D13) blink.

 [Open code](#)

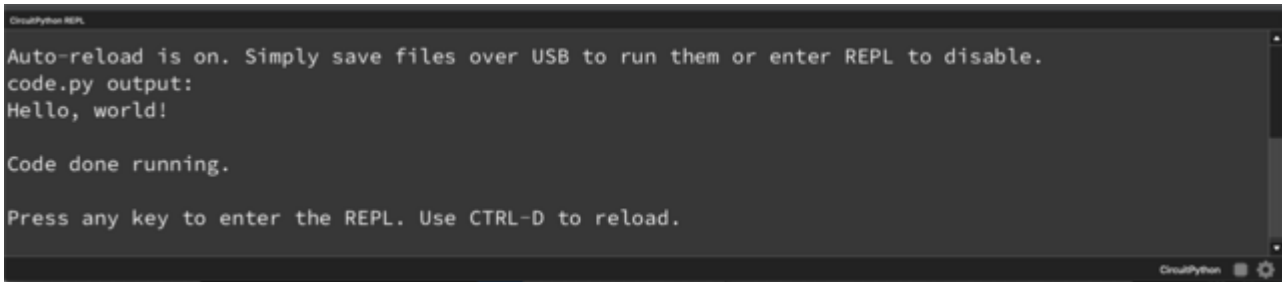
led.py

```
# Include all libraries
import time
import board
from digitalio import DigitalInOut, Direction

# Initialise LED, declare it an output
led = DigitalInOut(board.LED)
# led = DigitalInOut(board.D13) # Alternatively use the well-known pin 13
led.direction = Direction.OUTPUT

# Every second, flash the LED
while True:
    led.value = 0
    time.sleep(1)
    led.value = 1
    time.sleep(1)
```

Copy or write the above code into the Code with Mu editor and click the Save button to save it to the TinyML development board. The red LED should start to blink. Contrary to normal Python, CircuitPython requires the main code file to be named `code.py` and be located in the root of the `CIRCUITPY` drive. You can still work with multiple code files, just make sure that the main file is as described.

A screenshot of the CircuitPython REPL window. The window has a dark background with light-colored text. The text inside the window reads: "Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.", "code.py output:", "Hello, world!", "Code done running.", and "Press any key to enter the REPL. Use CTRL-D to reload." The window title bar at the top says "CircuitPython REPL". At the bottom right, there is a small "CircuitPython" logo and a settings gear icon.

```
CircuitPython REPL
Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.
code.py output:
Hello, world!

Code done running.

Press any key to enter the REPL. Use CTRL-D to reload.
```

If code outputs something (e.g. has a `print("Hello world!")` statement), the serial console can be used. Through the USB connection, it is possible to read out messages from the `print` statements using a serial console. To open the Serial console in Code with Mu, click the Serial button at the top of the window. The serial console will now show at the bottom of the window. For more information, take a look at the [Adafruit CircuitPython - Serial console page](#).

To check out more simple code examples, take a look at the [Adafruit Learning System - Guide to CircuitPython](#) or at the sensor code examples in the [previous section](#).

To access inputs and outputs (I/O) of the TinyML development kit, CircuitPython has defined easy references to the default pins. Certain inputs and outputs can be accessed using their semantic definition (e.g. the output connected to the red LED is `GPIO13`, however it can be referenced as `board.LED` in the code). All pin definitions can be found in the info box below. To learn more about the board library and pin usage, visit [CircuitPython - Pins and Board](#).

Pin definitions

The TinyML development kit has the following pin definitions and names:

Sensors

```
board.GPI014 board.D14 board.IO14 board.LD02
board.GPI01 board.D1 board.IO1 board.IR board.IR_RECV
board.GPI04 board.D4 board.I2S_CLK board.IO4 board.MIC_CLK
board.GPI05 board.D5 board.I2S_WS board.IO5 board.MIC_WS
board.GPI015 board.D15 board.I2S_DIN board.IO15 board.MIC_DIN
board.GPI02 board.D2 board.HALL board.HALL_EFFECT board.IO2
board.GPI021 board.D21 board.IMU board.IMU_INTERRUPT board.IMU_ISR board.IMU_WAKE board.IO21
```

Data busses

```
board.GPI018 board.D18 board.I2C1_SCL board.I2C_SCL board.IO18 board.SCL
board.GPI08 board.D8 board.I2C1_SDA board.I2C_SDA board.IO8 board.SDA
board.GPI016 board.D16 board.I2C2_SCL board.I2C_SCL2 board.IO16 board.SCL2
board.GPI017 board.D17 board.I2C2_SDA board.I2C_SDA2 board.IO17 board.SDA2
board.GPI043 board.D43 board.IO43 board.TX
board.GPI044 board.D44 board.IO44 board.RX
```

Expansion header

```
board.GPI06 board.ANALOG1 board.D6 board.IO6
board.GPI07 board.ANALOG2 board.D7 board.IO7
board.GPI09 board.ANALOG3 board.D9 board.IO9
board.GPI10 board.D10 board.IO10
board.GPI11 board.D11 board.IO11
board.GPI12 board.D12 board.IO12
board.GPI047 board.D47 board.IO47
board.GPI048 board.D48 board.IO48
```

Buttons

```
board.GPI00 board.BUTTON1 board.D0 board.IO0 board.BUTTON
board.GPI038 board.BUTTON2 board.D38 board.IO38
```

LEDs

```
board.GPI013 board.D13 board.IO13 board.LED board.STATUS
board.GPI039 board.D39 board.IO39 board.NEOPIXEL
```

The original I/O pins of the kit can also be found in the electronic schematic of the TinyML development kit, in the [previous section](#).

To see this overview of pin naming on the TinyML development kit, run the following code.



pin_mapping.py

```
# Include the necessary pins
import microcontroller
import board

# Start an empty array for storing the found pins
board_pins = []

# Loop through all pins in the microcontroller directory
for pin in dir(microcontroller.pin):

    # Check if a pin has been seen before
    if isinstance(getattr(microcontroller.pin, pin), microcontroller.Pin):
        pins = []

        # Add the pin to aliases if found before
        for alias in dir(board):
            if getattr(board, alias) is getattr(microcontroller.pin, pin):
                pins.append("board.{}".format(alias))
        if len(pins) > 0:
            board_pins.append(" ".join(pins))

# Print all pins
for pins in sorted(board_pins):
    print(pins)
```

As mentioned before, it is possible to use ready-made code libraries, for example to easily integrate sensors, outputs or connectivity. Some libraries are built-in to the CircuitPython firmware, others may need to be downloaded and included in the `lib` folder on the `CIRCUITPY` drive. The examples on the TinySpark platform only use built-in libraries, as all sensor libraries for the on-board sensors are included in CircuitPython. If you want to learn more about external libraries, visit [CircuitPython - Libraries](#).

Included libraries

The following libraries are built-in to the default CircuitPython installation on the TinyML development kit

- `analogio` (for analog I/O)
- `array` (for generating array objects)
- `bitbangio` (for simulating different communication protocols)
- `board` (for getting pin numbers and descriptions)
- `busio` (for using default data busses like I2C)
- `collections` (for special data containers)
- `digitalio` (for digital I/O)
- `math` (for basic mathematic operations and constants)
- `os` (for accessing operating level functions)
- `pwmio` (for controlling PWM devices)
- `random` (for generating random numbers)
- `register` (for attributes on data busses like I2C)
- `struct` (for defining data structures)
- `sys` (for accessing system and program functions)
- `time` (for timing)
- `ulab` (for Python NumPy like maths and variables)
- `usb_cdc` (for the USB connection)

The following libraries are used for the sensors and outputs on the TinyML development kit.

- `apds9930` (for the on-board Light and Distance sensor)
- `bme280` (for the on-board Environmental sensor)
- `ir_remote` (for the on-board IR receiver)
- `neopixel` (for the on-board Neopixels)
- `lsm6ds` (for the on-board Inertial motion sensor)

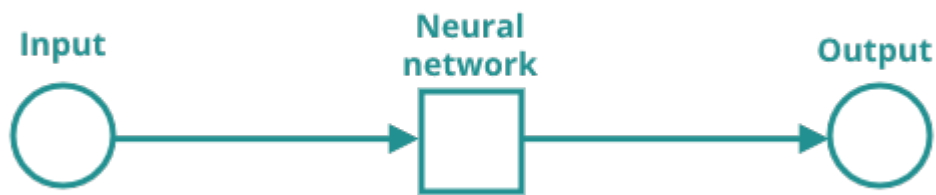
That's it, now you know everything needed to get started with the TinySpark TinyML material.

Get started on Chapter 1

3. Chapter 1

3.1 Chapter 1 - Introduction to Machine Learning

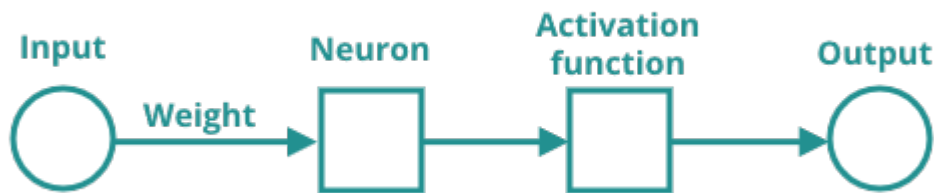
A neural network is a computational model that mimics the structure and function of biological neurons. A neuron is a basic unit of a neural network that can receive, process and transmit information. In this chapter, neuron functions and connections will be explained. After the explanation, a simple project will be programmed to show how a neuron functions mathematically.



In the next section, the basics of neurons are introduced.

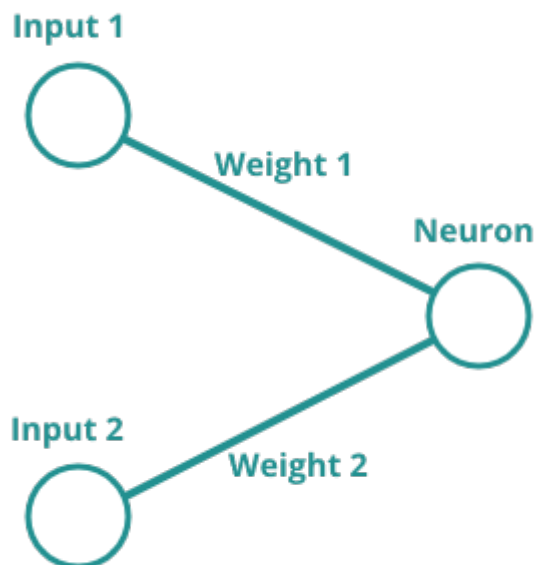
3.2 The Neuron

The neuron is at the heart of every neural network, it provides all computation and links to other neurons and layers within the neural network. A neuron consists of three main components: an input layer, an activation function and an output layer. The input layer receives signals from other neurons or external sources, such as images, texts, or numbers. The activation function determines whether the neuron should fire or not, based on the input signals. The output layer sends the firing signal to other neurons or to the final output of the network.



The inputs of a neuron come in many different shapes and sizes (literally). A neuron could receive just one single input, or be connected to more than 500 different inputs. The inputs to a neuron are always numeric (since this is essentially one giant mathematical formula), so complex inputs like sound or images have to be split into many different parts (e.g. milliseconds of sound or pixels in an image). These can then be fed into the input of a network.

The neuron with two inputs, two weights and a simple threshold activation function below will be examined. How do inputs travel through it and how do they lead to an output?



The first step in computation of the output is to sum the inputs of the neuron together with the weights of the neuron:

$$\sum \text{inputs} * \text{weights} = \text{input 1} * \text{weight 1} + \text{input 2} * \text{weight 2}$$

The activation function which comes after the inputs can be a simple threshold function that fires if the input signals exceed a certain value, or a more complex function that can capture nonlinear relationships between inputs and outputs. Some common activation functions are sigmoid, tanh, ReLU and softmax. More information can be found in the [Machine Learning glossary](#).

Now, the neuron needs to be *activated* by running the sum through the chosen activation function. In this example, a simple threshold function is chosen. If the sum of the neuron is ≥ 0.5 or higher, the output will be 1, otherwise the output will be 0.

$$f(x) = \begin{cases} 0 & \text{if } x < 0.5 \\ 1 & \text{if } x \geq 0.5 \end{cases}$$

Now, the output of the neuron with these inputs and weights is calculated as follows:

$$\begin{aligned} \text{input 1} &= 0.2 \quad \text{input 2} = 0.8 \quad \text{weight 1} = 0.3 \quad \text{weight 2} = 0.9 \\ \sum \text{inputs} * \text{weights} &= 0.2 * 0.3 + 0.8 * 0.9 = 0.78 \\ \text{output} &= f(0.78) = 1 \end{aligned}$$

The output of the neuron, with the given inputs would be 1.

If this is now implemented into a simple Python script, it could look like this:

Implementing this simple neuron logic into Python should be straightforward.



single_neuron.py

```
input1 = 0.2
input2 = 0.8
weight1 = 0.3
weight2 = 0.9

sum = (input1 * weight1) + (input2 * weight2)
if sum >= 0.5:
    activation = 1
else:
    activation = 0

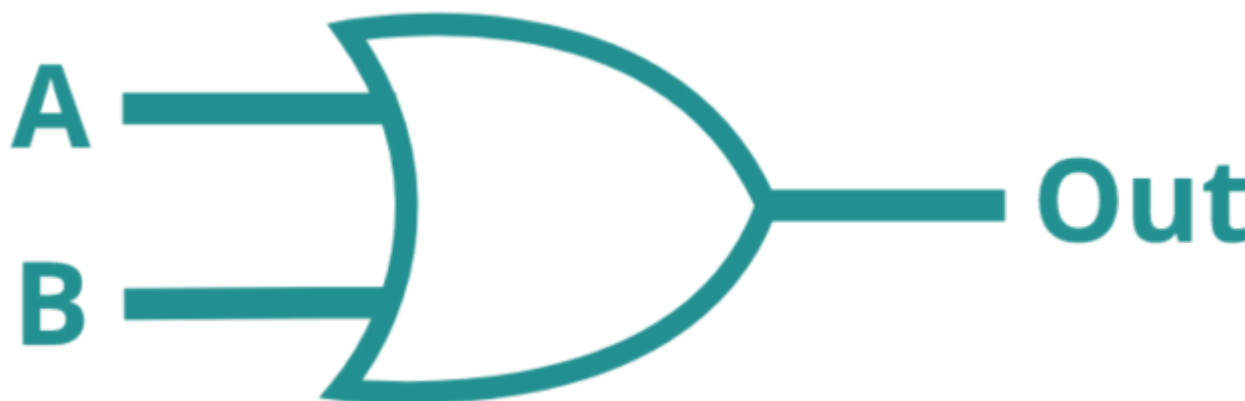
print(activation)
```

In the next section, the neuron that was just created will be used to predict something more meaningful.

3.3 Logic gates

A logic gate (in electronics), is a device which performs logical operations using binary logic. Logic gates are the fundamentals of modern processors, enabling chips to store data using Latches¹, perform binary calculations using Arithmetic Logic Units² and much more.

In this first Mini-project, a single-neuron 'network' will be built to replicate the behaviour of an OR logic gate³. The OR gate will activate whenever either of its inputs reads HIGH (or 1 in this case).



The logic table for an OR gate looks like this:

Input		Output
0	0	0
0	1	1
1	0	1
1	1	1

In order to make writing the equations easier, the sum and activation function are combined into one formula.

$$\text{output} = f(\text{sum of inputs} * \text{weights})$$

Now try if the neuron from the [previous section](#) will predict the correct output for the OR logic gate.

$$\begin{aligned} \text{input 1} &= 0 \quad \text{input 2} = 0 \quad \text{weight 1} = 0.3 \quad \text{weight 2} = 0.9 \\ \text{output} &= f(0 * 0.3 + 0 * 0.9) = 0 \end{aligned}$$

That looks promising, so the other inputs are tried as well.

$$\begin{aligned} \text{output} &= f(0 * 0.3 + 1 * 0.9) = 1 \quad \text{output} = f(1 * 0.3 + 0 * 0.9) = 0 \quad \text{output} = f(1 * 0.3 + 1 * 0.9) = 1 \end{aligned}$$

Almost correct, however with the inputs (1,0), the neuron wrongly outputs (0), which should be (1) (according to the logic table above).

How can this behaviour be changed?

By changing the weights.

Take a closer look at the calculation of the incorrect output.

$$\text{sum} = \text{input 1} * \text{weight 1} + \text{input 2} * \text{weight 2} = 1 * 0.3 + 0 * 0.9 = 0.3$$

Indeed, if the sum is run through the activation function, the result is incorrect.

$f(0.3)=0$

The output of the sum, 0.3 , is too low to trigger the activation function. By intuitively approaching the calculations above, there are some things that can be concluded:

- the inputs cannot be changed
- the activation function could be changed
- the weights could be changed

In neural networks, inputs cannot change, so it does not make sense to focus on that aspect of the calculation. The activation function could be changed, however since the activation function produced correct outputs for the other inputs, this is probably not the issue (additionally, activation functions in neural networks are only rarely changed to get to the desired output). Lastly, the weights can be changed, and consequently, this is the usual way of tuning a neural network to achieve correct outputs.

In order to trigger the activation function $f(x)$, the sum for inputs $(1,0)$ needs to be higher than or equal to 0.5 . If the $\text{weight } 1$ is increased to 0.5 for example, recalculate all outputs and see if the neuron 'network' is correct for all inputs.

$$\begin{aligned} \text{output} &= f(0*0.5 + 0*0.9) = 0 \\ \text{output} &= f(0*0.5 + 1*0.9) = 1 \\ \text{output} &= f(1*0.5 + 0*0.9) = 1 \\ \text{output} &= f(1*0.5 + 1*0.9) = 1 \end{aligned}$$

Alternatively, to see the changes $\text{weight } 1$ has on the outputs of the model, use the interactive visualisation below.

Weight 1 0.3

Now that the weights are tuned correctly, program this neuron 'network' into a simple Python script. To make sure that all possible inputs of the OR-gate are covered, they are stored inside of an array. The array will be looped over, and outputs for every input combination will be created.



single_neuron_OR_gate.py

```
inputs = [
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
]
weight1 = 0.5
weight2 = 0.9

for input in inputs:
    sum = (input[0] * weight1) + (input[1] * weight2)
    if sum >= 0.5:
        activation = 1
    else:
        activation = 0
    print(input, activation)
```

In the next section, this network will be deployed to the TinySpark development board, in order to experience how to expand the neuron 'network' beyond the screen.

1. [https://en.wikipedia.org/wiki/Flip-flop_\(electronics\)](https://en.wikipedia.org/wiki/Flip-flop_(electronics)) ↩
2. https://en.wikipedia.org/wiki/Arithmetic_logic_unit ↩
3. https://en.wikipedia.org/wiki/OR_gate ↩

3.4 Logic gates on the TinySpark development board

Now that the first neuron 'network' was successfully implemented, it will be transported to the TinySpark development kit.

The two buttons, `Button 1` and `Button 2` on the development kit will be used to simulate the inputs, and the LED (`LED13`) will be used to show if the output of the neuron 'network' is LOW/0 or HIGH/1. In order to access the Inputs and Outputs of the development kit, some code is needed. The code below combines the example code for interaction with the `buttons` and output through the `LED`.

 [Open code](#)

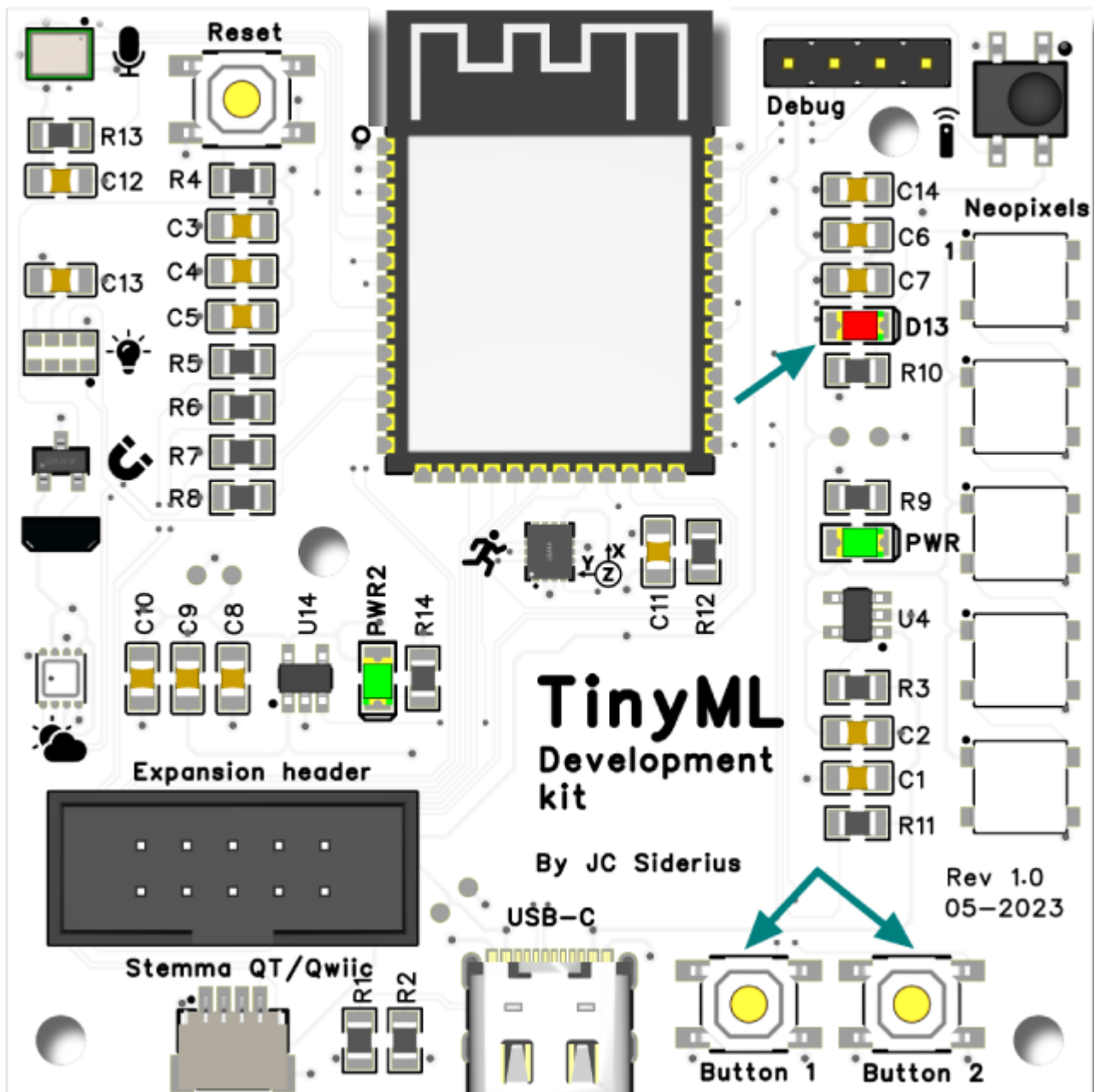
input_output.py

```
# import the library to take care of the pins
import board
from digitalio import DigitalInOut, Direction, Pull

# initialise the pins
button1 = DigitalInOut(board.BUTTON1)
button1.direction = Direction.INPUT
button1.pull = Pull.UP
button2 = DigitalInOut(board.BUTTON2)
button2.direction = Direction.INPUT
button2.pull = Pull.UP
led = DigitalInOut(board.LED)
led.direction = Direction.OUTPUT

# turn the LED on
led.value = True

# loop endlessly
while 1:
    # check if a button is pressed, and print if it is (buttons are pulled HIGH, so check for low)
    if not button1.value:
        print("button1 pressed")
    if not button2.value:
        print("button2 pressed")
```



Now the logic from the [previous section](#) needs to be implemented, in order to complete the neuron 'network' and successfully deploy it to the TinySpark development board.

[Github](#) [Open code](#)

OR_gate.py

```
# import the library to take care of the pins
import board
from digitalio import DigitalInOut, Direction, Pull

# initialise the pins
button1 = DigitalInOut(board.BUTTON1)
button1.direction = Direction.INPUT
button1.pull = Pull.UP
button2 = DigitalInOut(board.BUTTON2)
button2.direction = Direction.INPUT
button2.pull = Pull.UP
led = DigitalInOut(board.LED)
led.direction = Direction.OUTPUT

# store the weights
weight1 = 0.5
weight2 = 0.9
```

```
# loop endlessly
while 1:
    # Read the button value (buttons are pulled HIGH, so check for low)
    input1 = button1.value
    input2 = button2.value

    sum = (input1 * weight1) + (input2 * weight2)
    if sum >= 0.5:
        # activation = 1
        led.value = True
    else:
        # activation = 0
        led.value = False
```

Upload the code and see if the neuron 'network' functions as expected, by pressing `Button 1` and `Button 2` and observing the LED.

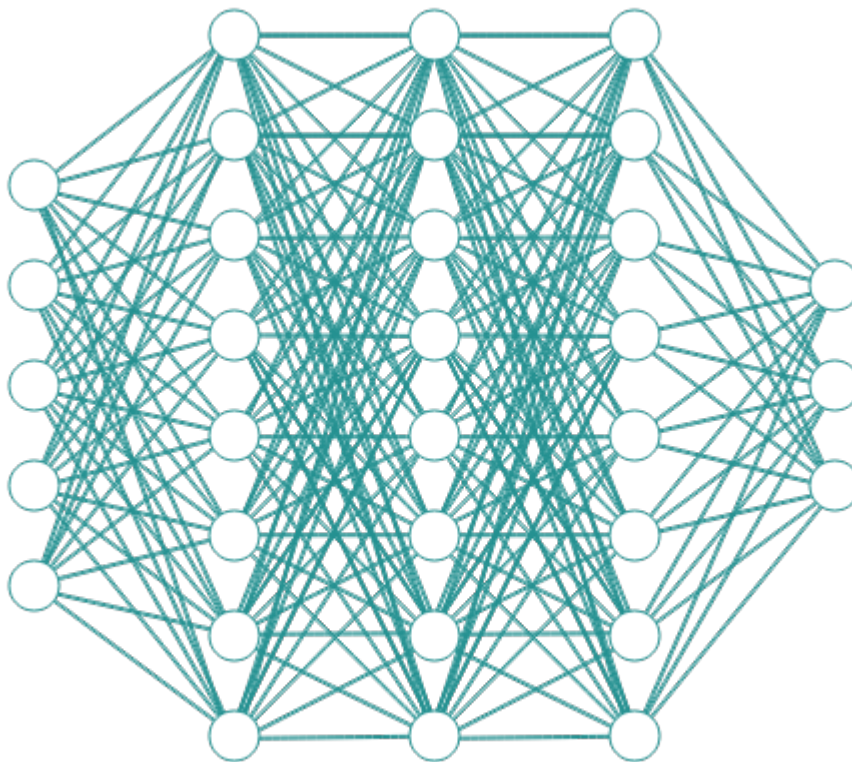
In the next chapter, some more complicated predictions will be made by utilising an actual network of neurons.

4. Chapter 2

4.1 Chapter 2 - The neuron problem

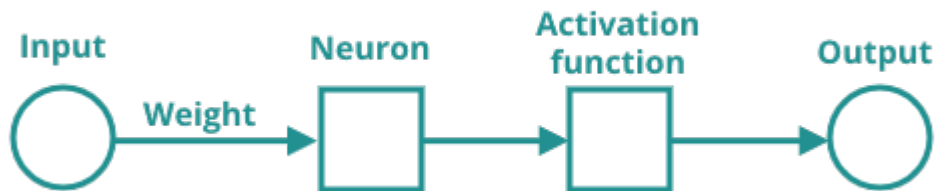
As shown in the last chapter, a neuron is capable of predicting outcomes based on the inputs and weights it is given. A single neuron can only perform linear or simple non-linear transformations, which may not be sufficient to represent the complexity of real-world problems. To overcome this restriction, a network of neurons can model non-linear relationships better than a single neuron, because it can capture more features and interactions among the inputs.

The network component of a neural network is the structure that defines how the neurons are arranged and connected. A neural network consists of multiple layers of neurons, each layer performing some computation on the inputs from the previous layer and passing the outputs to the next layer. The first layer is called the input layer, and the last layer is called the output layer. The layers in between are called hidden layers. The number and size of the hidden layers determine the complexity and capacity of the neural network. In this chapter, a closer look will be taken at the network that forms prediction models.

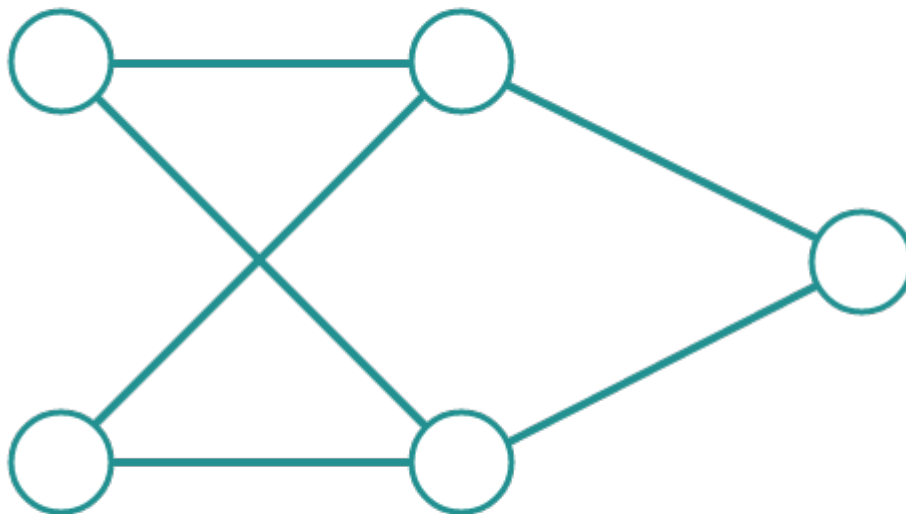


4.2 Network connections

Looking back at a single neuron, which has an input, weight, activation function and output, it is easy to see how one could chain neurons together to form a network.



By connecting the output of a neuron to the input of another neuron, the beginning of a network is formed. Extending this to more neurons, for example, two input neurons, two hidden neurons and one output neuron. If all neurons are interconnected to each other, like in the image below, this is called a Fully-connected Neural Network.



Input neurons or nodes generally have no activation function and weights associated to them, they merely provide a place to store inputs for further computation.

In the next section, the Fully-connected Neural Network in the image above will be implemented in Python.

4.3 Plant monitoring

In the first chapter, Logic gates were introduced. The mini-project programmed an OR gate in Python, and later implemented it onto the TinySpark development board.

During this mini-project, plant care is of highest regard. The *Fictioplantus*¹, as can be read below, is a very delicate plant from the Botanica Kingdom. It requires careful control of both temperature and humidity, as it will grow very poorly when conditions are not right.

Fictioplantus

Fictiocactus maximus


Origin: Botanica Kingdom
Humidity Threshold: 70%
Temperature Threshold: 25°C

Description: The Fictioplantus (Fictiocactus maximus), is an extraordinary plant native to the whimsical Botanica Kingdom. It exhibits unique growth patterns influenced by its surrounding environment, particularly humidity and temperature levels.

Plant Health Criteria:

Ideal Conditions: When both humidity and temperature are below or above the specified thresholds, the Fictioplantus grows vigorously and flourishes with lush foliage.

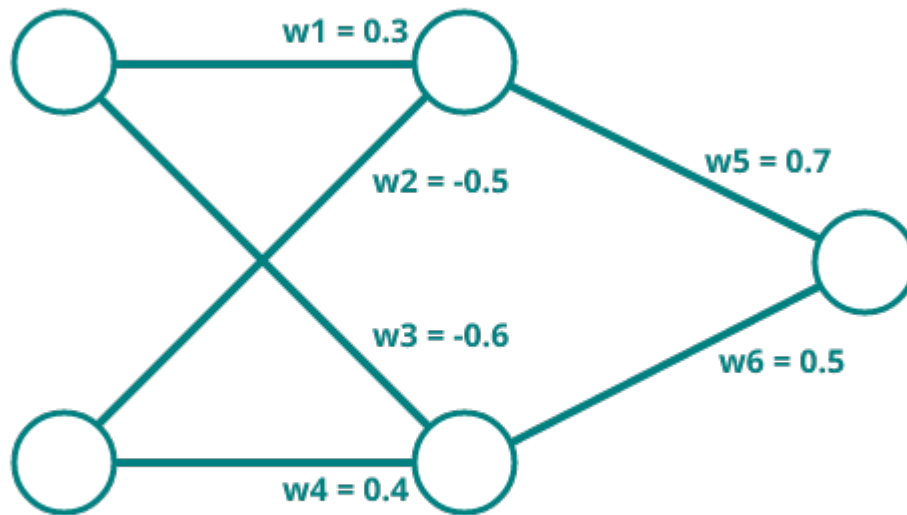
Poor Growth: If either humidity or temperature exceeds the threshold while the other remains within the ideal range, the Fictioplantus experiences poor growth and may exhibit stunted development, wilting, or discoloration.



The requirements for succesful growth are very particular, they are summarized in the table below.

Temperature	Humidity	Growth
below 25°C	below 70%	no
above 25°C	above 70%	no
below 25°C	above 70%	yes
above 25°C	below 70%	yes

Classifying inputs like the temperature and humidity is more difficult than classifying simple logic gates. The reason behind this is the case of linear and non-linear separability². Separability refers to the property of a dataset or set of points (in this case the inputs) where it is possible to draw a straight line that can completely separate the points into different classes (below/above 25° or below/above 70% humidity in this case). The problem proposed above is such a non-linearly separable problem. To overcome this, it is nescessary to introduce more neurons into the neural network.



The network from the [previous section](#) is used here. Weights are defined as seen. For the activation function, the step function from [Chapter 1](#) is used again. Note how in this example, negative weights are also possible in order to give inputs negative influences as well.

```
\[ f(x) = \begin{cases} 0 & \text{if } x < 0.5 \\ 1 & \text{if } x \geq 0.5 \end{cases} \]
```

```
\[ \displaylines{ \text{weight 1}=0.3 \text{weight 2}=-0.5 \text{weight 3}=-0.6 \text{weight 4}=0.4 \text{weight 5}=0.7 \text{weight 6}=0.5 } \]
```

In order to process the inputs correctly, some pre-processing of the measurements needs to take place. This is done in order to make calculations easier, and to keep weights manageable (in this case within ranges -1 and 1). The calculation is described below. In complicated neural networks, this pre-processing can be part of the network, in a so called Convolution step³.

```
\[ \displaylines{ temp_{in} = (temperature - 25) / 10 \quad humid_{in} = (humidity - 70) / 10 } \]
```

Calculating the outputs for some possible input combinations is now more complicated than in the previous chapter. As inputs, a few values above and below the threshold are chosen: $(22)^{\circ}\text{C}$ and $(30)^{\circ}\text{C}$ for temperature, $(40)\%$ and $(85)\%$ for humidity. In the calculations below, first the pre-processing of the values is done, then the respective outputs are calculated. Remember that because the activation function used is still a step function, the output will always be either (0) or (1) (signifying poor and good growing conditions respectively).

```
\[ \displaylines{ \text{input}(22^{\circ}\text{C})=(22 - 25) / 10 = -0.3 \text{input}(30^{\circ}\text{C})=(30 - 25) / 10 = 0.5 \text{input}(40\%)=(40 - 70) / 10 = -3 \text{input}(80\%)=(80 - 70) / 10 = 1 } \]
```

Network calculation results

For the weights above, the calculated predictions for the network can be found below.

$$\begin{aligned} \text{output}(-0.3, -3) &= f(f(-0.3 \cdot 0.3 + -3 \cdot -0.5) \cdot 0.7 + f(-0.3 \cdot -0.6 + -3 \cdot 0.4) \cdot 0.5) = 0 \\ \text{output}(-0.3, 1) &= f(f(-0.3 \cdot 0.3 + 1 \cdot -0.5) \cdot 0.7 + f(-0.3 \cdot -0.6 + 1 \cdot 0.4) \cdot 0.5) = 1 \\ \text{output}(0.5, -3) &= f(f(0.5 \cdot 0.3 + -3 \cdot -0.5) \cdot 0.7 + f(0.5 \cdot -0.6 + -3 \cdot 0.4) \cdot 0.5) = 1 \\ \text{output}(0.5, 1) &= f(f(0.5 \cdot 0.3 + 1 \cdot -0.5) \cdot 0.7 + f(0.5 \cdot -0.6 + 1 \cdot 0.4) \cdot 0.5) = 0 \end{aligned}$$

Using the interactive visualisation below, try to tune the weights so that the prediction is correct for the given inputs. The weights given above can be used as guidance, however there are many different possible combinations of weights to be found that will lead to the desired output. *Note that this tuning is meant more as an exercise to see what effects weights have in a neural network. It is expected to be very difficult to find correct weights by hand.*

Weight 1 0

Weight 2 0

Weight 3 0

Weight 4 0

Weight 5 0

Weight 6 0

Temperature 23

Humidity 40

Now program the found weights (of the pre-given ones) into a simple Python script. The weights of the network will be stored inside of an array. The inputs for temperature and humidity can be either input manually, or fetched from an external API⁴ that supplies weather data, such as the Dutch weather forecast [Buienradar API](#).

 Open in Colab

plant_monitoring.py

```
# import the libraries to fetch weather data from an API
import requests
import json

# store the weights
weights = [
    0.3,
    -0.5,
    -0.6,
    0.4,
    0.7,
    0.5
]

# defining the activation function
def activation(x):
    if x >= 0.5:
        return 1
    else:
        return 0

# formulate the API request
response = requests.get('https://data.buienradar.nl/2.0/feed/json')

# get the temperature and humidity from the Buienradar API
# the location is currently set to De Bilt in the Netherlands
temperature = response.json()['actual']['stationmeasurements'][3]['temperature']
humidity = response.json()['actual']['stationmeasurements'][3]['humidity']

# Alternatively, input the temperature and humidity manually
# temperature = 23
# humidity = 65

# Print the inputs
print(f"Temperature: {temperature}°C, Humidity: {humidity}%")

# pre-processing the inputs
temp_in = (temperature - 25) / 10
humid_in = (humidity - 70) / 10

# perform network calculations
neuron1 = activation( (temp_in * weights[0]) + (humid_in * weights[1]) )
neuron2 = activation( (temp_in * weights[2]) + (humid_in * weights[3]) )
output = activation( (neuron1 * weights[4]) + (neuron2 * weights[5]) )
```

```
# printing the result
if output == 1:
    print("Growing conditions are good")
else:
    print("Growing conditions are poor")
```

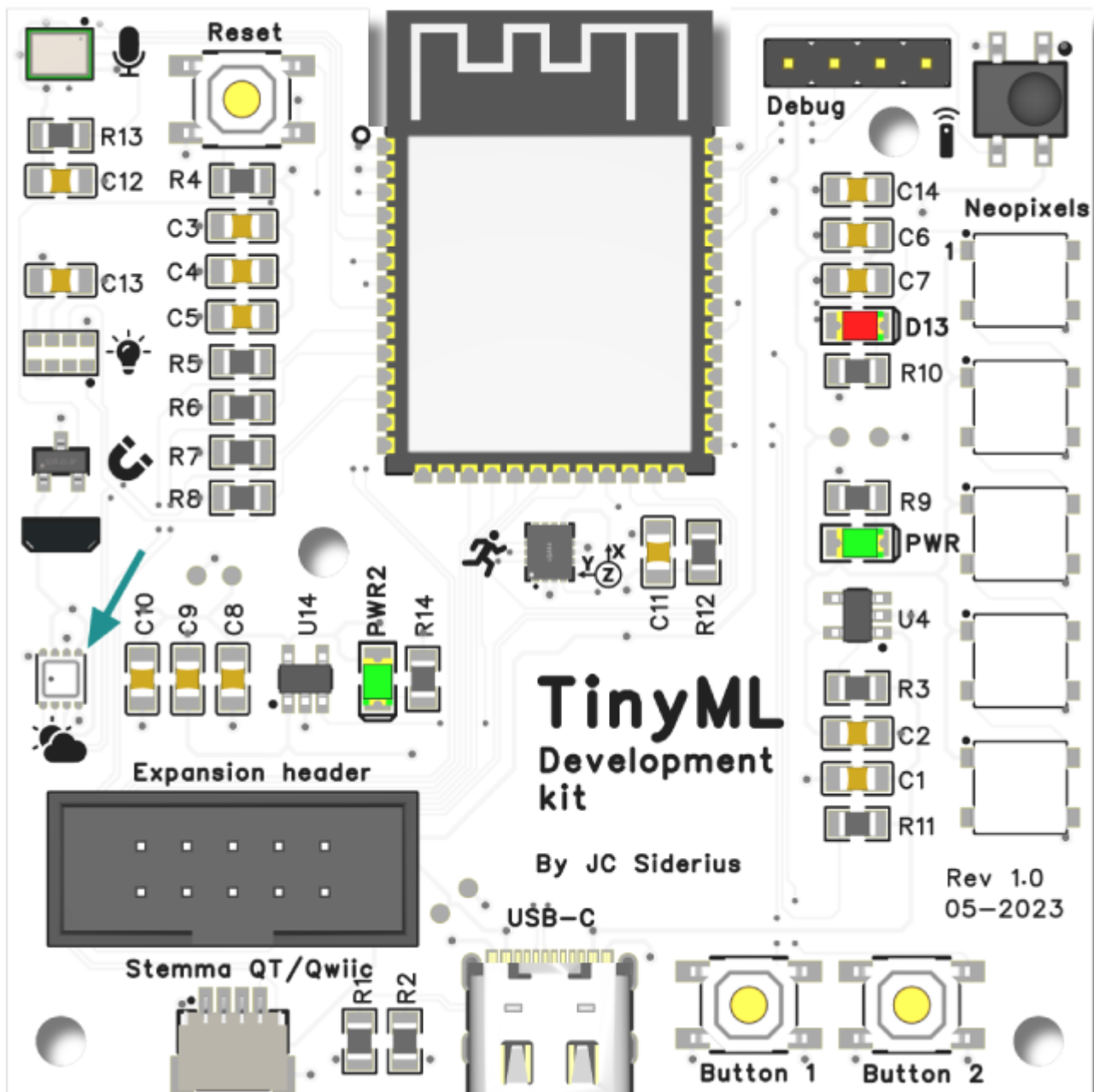
In the next section, the network will be deployed to the TinySpark development board, utilising the on-board environmental sensor.

1. The Fictioplantus acts as a simplified plant example in this case, although principles learned in this project can be applied to real plant monitoring. ↩
2. https://en.wikipedia.org/wiki/Linear_separability ↩
3. [https://en.wikipedia.org/wiki/Layer_\(deep_learning\)](https://en.wikipedia.org/wiki/Layer_(deep_learning)) ↩
4. <https://en.wikipedia.org/wiki/API> ↩

4.4 Plant monitoring on the TinySpark development board

Now that you have implemented your first real neural network, deploy the model to the TinySpark development kit.

The environmental sensor (more information on the [TinyML development kit sensors page](#)) will be used for the temperature and humidity measurements. In a deployment, this sensor could be mounted close to the plant, and be configured to send a message to the gardener, or remotely open a window or start a ventilator. For now, the result of the prediction will be printed to the Serial monitor (see the [Programming guide](#) for more information).



Implement the logic from the [previous section](#), into the TinySpark development board, using the [Environmental sensor example code](#).

[Github](#) [Open code](#)

`plant_monitoring.py`

```

# import the libraries to take care of the sensor and manage time
import board
import time
from adafruit_bme280 import basic as adafruit_bme280

# initialise the environmental sensor
i2c = board.I2C()
bme280 = adafruit_bme280.Adafruit_BME280_I2C(i2c, address=0x76)

# store the weights
weights = [
    0.3,
    -0.5,
    -0.6,
    0.4,
    0.7,
    0.5
]

# defining the activation function
def activation(x):
    if x >= 0.5:
        return 1
    else:
        return 0

# loop endlessly
while 1:
    # get the sensor readings and print them
    temperature = bme280.temperature
    humidity = bme280.relative_humidity
    print(f"Temperature: {temperature}, Humidity: {humidity}")

    # pre-processing the inputs
    temp_in = (temperature - 25) / 10
    humid_in = (humidity - 70) / 10

    # perform network calculations
    neuron1 = activation( (temp_in * weights[0]) + (humid_in * weights[1]) )
    neuron2 = activation( (temp_in * weights[2]) + (humid_in * weights[3]) )
    output = activation( (neuron1 * weights[4]) + (neuron2 * weights[5]) )

    # printing the result
    if output == 1:
        print("Growing conditions are good")
    else:
        print("Growing conditions are poor")

    # perform the prediction every 2 seconds to see if it has changed
    time.sleep(2)

```

Now upload the code and see if the network works on the development board. You can try to manipulate the readings for temperature and humidity by breathing or blowing on the environmental sensor (see where it is on the board in the [TinyML kit introduction](#)). To cool the sensor down again or to get the humidity lower, let some ambient air go over the sensor, for example by holding it in the wind or by putting a fan over it.

In the next chapter, a neural network will be mathematically trained, and a program is written to repeat this many times, making it possible to create more accurate and larger networks.

5. Chapter 3

5.1 Chapter 3 - Training networks

The network shown in the last chapter was still relatively simple. Tuning the weights was still handled manually and by intuition. However, if networks become larger, tuning the optimal weights for each neuron by hand is impossible. Through the use of mathematics, it is possible to calculate the influence of every weight in the network systematically, and determine how to optimally change all weights to get the required output.

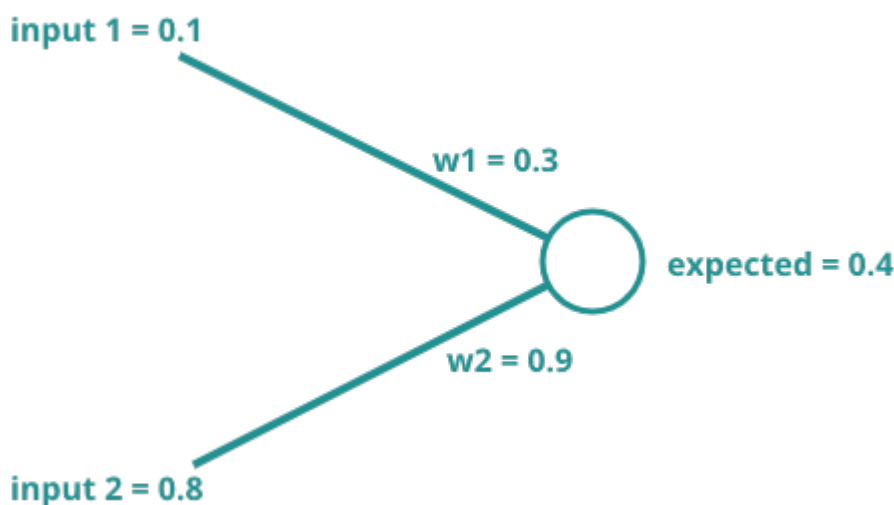
A walkthrough and explanation of the mathematics will be introduced first, then the calculations will be programmed so they can be performed automatically. Lastly, a model for classifying gestures will be trained using Python and subsequently deployed to the TinySpark development kit.

5.2 Training networks

In the last chapters, neuron weights were intuitively tuned to produce precise predictions. The mathematical approach to tuning weights will be explained more methodically.

To start off, a small recap to the calculation of the output of neurons. First, all inputs get summed together with their weights. Then, an activation function is applied to the result and this gives the final output of a neuron. This is commonly known as the *Feedforward* of a neural network (as in, feeding inputs forward through a neural network, until an output is generated).

In training a neuron, almost the opposite of this process is performed, in the so-called *Backpropagation*¹. To start, an input and the expected output is needed. For this example, a single neuron will be used to show all calculations, and in a later section, the mathematics for a simple network will be explained.



The neuron above has the following weights, inputs, expected output and new activation function (this is the linear activation function, chosen for it's simplicity).

```
\[ \displaylines{ \text{input1}=0.1\ \text{input2}=0.8\ \text{expected 1}=0.4\ \text{weight1}=0.3\ \text{weight2}=0.9\ } \]
```

```
\[ f(x) = x \]
```

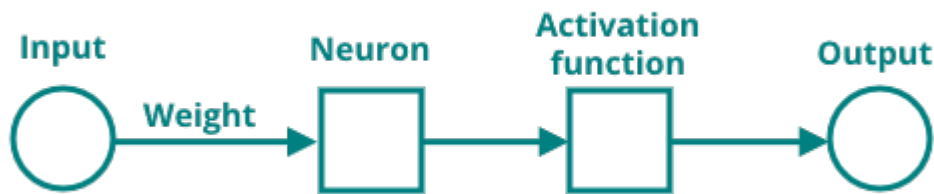
The feedforward of this neuron will look as follows.

```
\[ \displaylines{ \sum \text{inputs} * \text{weights} = 0.1 * 0.3 + 0.8 * 0.9 = 0.75\ \text{output}=f(0.75)=0.75\ } \]
```

As can be seen, the predicted result deviates a considerable amount. Mathematically speaking, this offset is the error of the prediction. It is calculated as follows.

```
\[ \displaylines{ \text{error}=\text{output}-\text{expected}\ \text{error}=0.75-0.4=0.35\ } \]
```

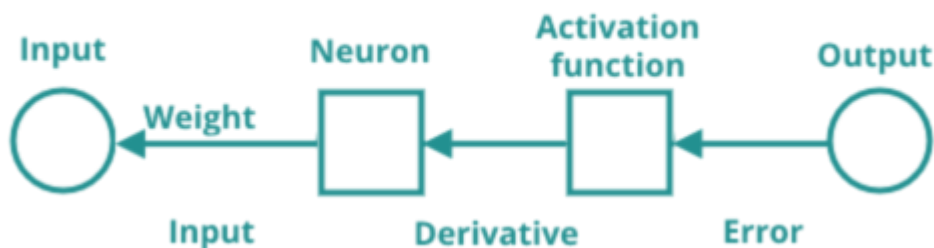
In order to now calculate the required changes to the weights of the network (called the delta / Δ), it is necessary to walk 'backwards' through the network, which in this example only consists of one neuron, and discover the influence of each weight on the final prediction. If the influence is then known, the appropriate changes can be applied to the weights. This technique is called Stochastic Gradient Descent², but for now, just focus on the application.



During the feedforward, two steps were performed:

1. The summation of inputs and weights
2. The activation of the sum

Additionally, one last step is performed at the output of the network, namely the calculation of the error of the prediction.



To calculate backwards, the derivatives of the mentioned steps need to be determined. For step 2, this requires the derivative of the activation function (in this case the linear function).

$$\text{f}(x)=x \quad \text{f}'(x)=1$$

For step 1, the derivate of the summation is also easy to calculate.

$$\text{sum} = \text{input1} * \text{weight1} + \text{input2} * \text{weight2} \quad \text{sum}'_{\text{weight1}} = \text{input1}$$

To calculate the total influence, or delta, all values need to be multiplied together. This is shown for both weights below.

$$\Delta_{\text{weight1}} = \text{error} * \text{f}'(x) * \text{sum}'_{\text{weight1}} = 0.35 * 1 * 0.1 = 0.035 \quad \Delta_{\text{weight2}} = \text{error} * \text{f}'(x) * \text{sum}'_{\text{weight2}} = 0.35 * 1 * 0.8 = 0.28$$

Now these deltas can be applied to the weights of the neuron. Note that the delta Δ is being subtracted from the weight, since there is an overshoot of the initial output.

$$\text{weight1}_{\text{new}} = \text{weight1} - \Delta_{\text{weight1}} = 0.3 - 0.035 = 0.265 \quad \text{weight2}_{\text{new}} = \text{weight2} - \Delta_{\text{weight2}} = 0.9 - 0.28 = 0.62$$

With these new weights, the output of the neuron has come closer to the expected output. To check if this is indeed the case, the error can be re-calculated as well.

$$\sum \text{inputs} * \text{weights}_{\text{new}} = 0.1 * 0.265 + 0.8 * 0.62 = 0.5225 \quad \text{output} = \text{f}(0.5225) = 0.5225 \quad \text{error} = 0.5225 - 0.4 = 0.1225$$

The neuron has now become more accurate at predicting the output. If the above steps are repeated more often, the network will slowly evolve and come closer to an error of 0 . In programming, the repeating training steps are often called *epochs*, with

training for a neural network sometimes needing just 100 epochs to get the desired accuracy, or sometimes requiring more than one million epochs.

This training can of course be programmed in Python.



training_example.py

```
# Define the inputs, expected output and weights
input1 = 0.1
input2 = 0.8
expected_output = 0.4

weight1 = 0.3
weight2 = 0.9

# Define the (simple) linear activation function
def activation(x):
    return x

# Calculate the output and error of the output
output = activation(input1 * weight1 + input2 * weight2)
error = output - expected_output

print(f"before training: {output=}, {error=}")

# Calculate the deltas for the weights
d_weight1 = 1 * input1 * error
d_weight2 = 1 * input2 * error

# Apply the deltas to the weights
weight1 = weight1 - d_weight1
weight2 = weight2 - d_weight2

# Calculate the output and error of the output again
output = activation(input1 * weight1 + input2 * weight2)
error = output - expected_output

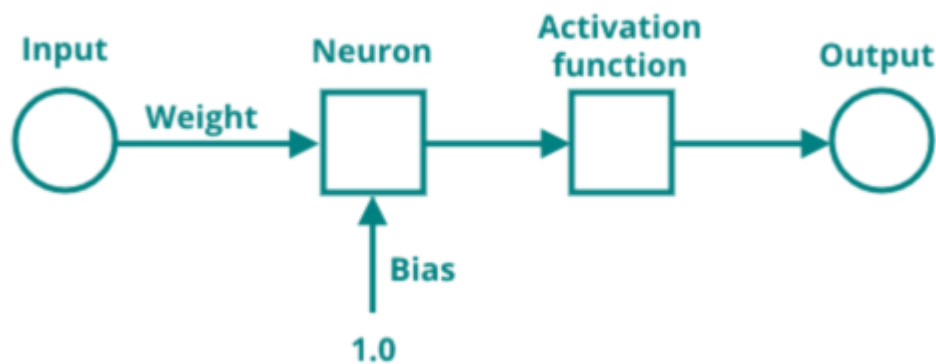
# Print the output and new weights
print(f"after training: {output=}, {error=}")
print(f"new weights: {weight1=}, {weight2=}")
```

In training more complicated neural networks, there are a few more nuances to keep in mind. First, in networks with more possible inputs and expected outputs, it is common to run each input through the training calculation and average all deltas for each respective weight before applying them to that weight. That way, no single input / expected output combination will have too much of an influence on the change in weights.

$$\Delta_{\text{weight}} = \frac{\Delta_1 + \Delta_2 + \dots + \Delta_n}{n}$$

Next, each neuron in a neural network often has an additional weight applied to its sum called the *bias*³. This bias, which has an input that is a constant (1), makes sure that the neuron it is attached to can produce a valid output even when all inputs are zeros. The summation function of a neuron is thus slightly changed as seen below.

$$\sum (\text{inputs} * \text{weights}) + \text{bias}$$



There is often also a *learning rate*⁴ applied to the delta before it changes the weight. This learning rate (common rates include 0.1, 0.05 or 0.01) limits the influence of a single change on the weight. The limiting is important to prevent overshooting the expected output(s) and generating weights that are so tiny or large that they have uncontrolled effects on the network.

$$\text{weight}_{\text{new}} = \text{learning rate} * \text{delta}_{\text{weight}}$$

Lastly, it can be beneficial to split input / expected output combinations in subsets, if there exist a lot of training data. This is called *mini-batching*⁵ and is done to reduce the overall computation- and memory needs to train the network one iteration / epoch. Additionally, it can give the network a faster training cycle as well as quicker approach to the desired accuracy of the model.

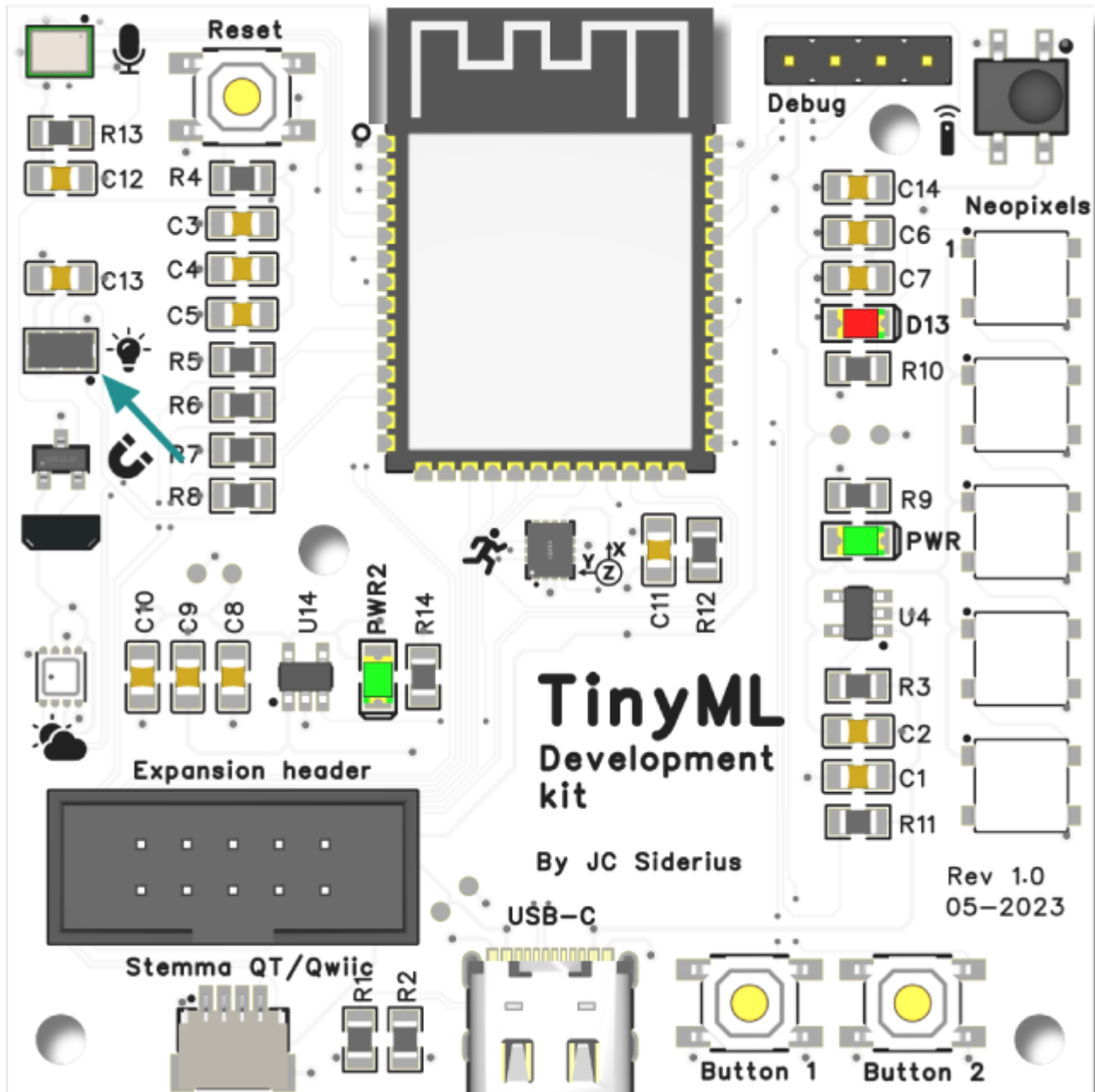
In the next section, the training of a neural network is programmed in Python, and an example network is trained using gesture data from the TinySpark development kit.

-
1. <https://en.wikipedia.org/wiki/Backpropagation> ↩
 2. https://en.wikipedia.org/wiki/Stochastic_gradient_descent ↩
 3. <https://machine-learning.paperspace.com/wiki/weights-and-biases> ↩
 4. https://en.wikipedia.org/wiki/Learning_rate ↩
 5. https://mzuer.github.io/machine_learning/mini_batch ↩

5.3 Gesture recognition - data acquisition

In this mini-project, a simple gesture recognition system will be built. Using the on-board proximity sensor introduced in the [TinySpark development section](#), two states will be detected:

- moving closer
- moving away



To detect these gestures, a network that takes in multiple proximity readings needs to be devised. In this example, three measurements will be input into the network: one measurement that is current, one that was 250ms ago, and one that is 500ms ago. This *Time series*¹ prediction method is a common way to analyse real-time sensor data using a neural network.

In order to train an accurate neural network, it is important to record actual data on the end device. In more complex systems however, it is not feasible to train the neural network on the data recording device, as it would take far too long to perform all training calculations there. Therefore, it is common practise to capture measurements on the local device, transfer them to a

more powerful computer (e.g. PC or even to the cloud) and perform the training there. Afterwards, only the tuned weights and the feedforward part of the neural network are transferred back to the local device in order to make predictions.



To start the mini-project, some data needs to be recorded on the TinySpark development kit. The code below will start the recording of the datapoints one second after `Button 1` is pressed. The LED will show when a recording is made. After the recording has finished, the values will be stored in an array. Once `Button 2` is pressed, all datapoints are printed onto the serial console. For information on using the serial console, please read the [TinySpark Programming section](#). From there, it is possible to copy the measurements over to a training program, which will be discussed further along this section.

 [Open code](#)

gesture_data_recording.py

```
# Import all libraries
import time
import board
from apds9930.apds9930 import APDS9930
from digitalio import DigitalInOut, Direction, Pull

# Initialize I2C
i2c = board.I2C()
sensor = APDS9930(i2c)

# Initialise buttons 1 and 2
button1 = DigitalInOut(board.BUTTON1)
button1.direction = Direction.INPUT
button1.pull = Pull.UP
button2 = DigitalInOut(board.BUTTON2)
button2.direction = Direction.INPUT
button2.pull = Pull.UP

# Initialise LED
led = DigitalInOut(board.LED)
led.direction = Direction.OUTPUT

# Array for storing the measurements
proximity_readings = list()

# Loop continuously
while True:

    # Check button 1 (note the signal is pulled up / high, so check for low signal)
    if button1.value == 0:

        # Startup delay
        recording = list()
        print("> Starting recording in 1sec")
        time.sleep(1)

        # Start recording
        led.value = 1
        distance = sensor.proximity
        recording.append(distance)
        print(f"{distance}")
        # delay for 250ms
        time.sleep(0.250)
        distance = sensor.proximity
        recording.append(distance)
        print(f"{distance}")
        # delay for another 250ms
        time.sleep(0.250)
        distance = sensor.proximity
        recording.append(distance)
        print(f"{distance}")
```

```

# Stop recording
led.value = 0
print("> Stopped recording")
proximity_readings.append(recording)

# Check button 2
elif button2.value == 0:

    # Print all recordings to the serial console, in form of an array
    print("Recordings: [-500ms, -250ms, current]")
    print("")
    for recording in proximity_readings:
        print(f"[{recording[0]}, {recording[1]}, {recording[2]}],")
    print("")

    # Clear the measurement storage
    proximity_readings = list()

    # Pause after resetting
    time.sleep(1)

# Debounce the buttons
time.sleep(0.01)

```

To record some datapoints, press **Button 1** and when the LED turns on, then perform a gesture. After recording some datapoints, press **Button 2** to print them to the serial console. It is advisable to record all datapoints for one type of gesture, then print those, then continue to record the next type. Store the measurements somewhere (as they will be used in the next section) and make sure to label them appropriately. *About 5-7 datapoints per gesture are enough for this example.*

Proximity sensor measurements

Since the proximity sensor included on the TinySpark development board was originally produced for use in hand-detecting applications such as automatic soap dispensers, the optimal measurement range is between 10-25cm from the sensor. Additionally, it is easiest to use a solid coloured object such as a piece of paper or cardboard to record the gestures, as this ensures accurate detection.

Use the distance printed to the serial console in order to see if the measurement(s) makes sense. If they don't seem right, try again and delete the old measurement(s) by emptying the array using **Button 2**. Also look at [TinySpark development kit description](#) to see the example code for the proximity (and light) sensor.

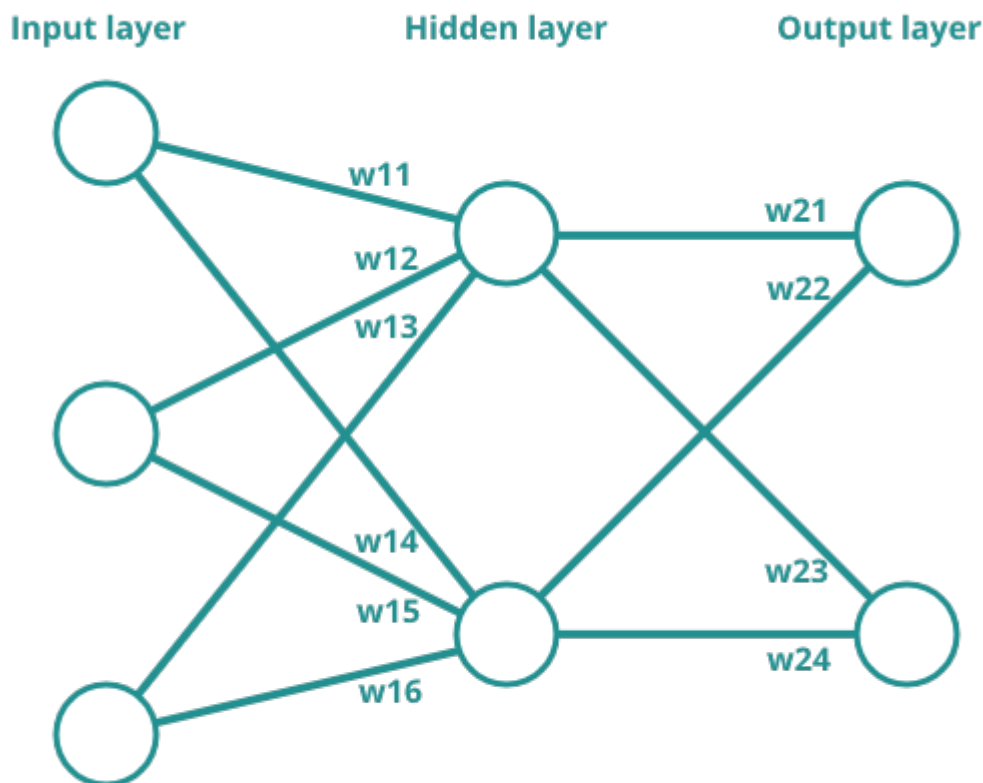
In the next section, the measurements will be run through the neural network and the weights will be trained using backpropagation.

1. https://en.wikipedia.org/wiki/Time_series ←

5.4 Gesture recognition - training the model

To continue building the neural network that will recognise gestures, the mathematics as discussed in a [previous section](#) will be implemented in Python. Then the aquired data from the [last section](#) will be used to train the network. Finally, the tuned weights will be used to run the model on the TinySpark development kit in order to detect gestures.

For this model, a neural network with two hidden neurons and two output neurons is chosen. Since three measurements are taken, three input neurons are required. For the output, two neurons are chosen, as to give the probability of each of the gestures: $\text{output1} = p(\text{moving closer})$ and $\text{output2} = p(\text{moving away})$. The weights are initialised randomly, since it is not possible to determine starting weights by hand in a meaningful way anymore. The randomisation uses a uniform random distribution between (-1) and (1) . Below, a depiction of the described neural network can be found.



With the measurements taken in the [previous section](#), one example calculation will be performed on a *moving closer* measurement. All variables that belong together will be grouped into an array for convenience. Additionally, the linear activation function is defined.

```
\[ \displaylines{ \text{measurement}=[5.1, 46.7, 120.5] \text{expected output}=[1, 0] \text{weights}_1=[-0.50, 0.22, -0.91, 0.61, 0.14, -0.48] \text{weights}_2=[0.73, 0.29, -0.29, -0.24] f(x)=x } \]
```

Since the measurement values are quite large, and it is best to keep the weights in a network within managable ranges (e.g. between (-2) and (2)), some pre-processing of the data needs to be performed again. For each measurement in the measurements array, the following calculation will be performed

```
\[ \text{measurement}_{in} = \frac{\text{measurement}}{100} \]
```

```
\[ \displaylines{ \text{measurement}=[5.1, 46.7, 120.5] \text{measurement}_{in}=[0.051, 0.467, 1.205] } \]
```

The hidden layer can now be determined by using the feedforward calculation.

$$\begin{aligned} \text{hidden}_1 &= f(\text{measurement}_1 * \text{weights}_{11} + \text{measurement}_2 * \text{weights}_{12} + \text{measurement}_3 * \text{weights}_{13}) \\ \text{hidden}_2 &= f(\text{measurement}_1 * \text{weights}_{14} + \text{measurement}_2 * \text{weights}_{15} + \text{measurement}_3 * \text{weights}_{16}) \\ \text{output}_1 &= f(\text{hidden}_1 * \text{weights}_{21} + \text{hidden}_2 * \text{weights}_{22}) \\ \text{output}_2 &= f(\text{hidden}_1 * \text{weights}_{23} + \text{hidden}_2 * \text{weights}_{24}) \end{aligned}$$

$$\text{hidden}_1 = -1.0193 \quad \text{hidden}_2 = -0.4819 \quad \text{output}_1 = -0.8839 \quad \text{output}_2 = 0.4112$$

Now the error is calculated. Additionally, since there is more than one output, the *Loss* is calculated. This serves as a measure of all error in the system (the compound error). There are many different loss functions¹, but for the sake of simplicity, the loss function will just be a summation of the errors.

$$\begin{aligned} \text{error}_1 &= \text{output}_1 - \text{expected output}_1 = -0.8839 - 1 = -1.8839 \\ \text{error}_2 &= \text{output}_2 - \text{expected output}_2 = 0.4112 - 0 = 0.4112 \\ \text{loss} &= \text{error}_1 + \text{error}_2 = -1.4723 \end{aligned}$$

To now make the model more accurate, backpropagation will be performed. First, the deltas of the output layer weights (weights_{21}) will be calculated. This is again done by going 'backwards' and using the derivatives, as shown in one of the [previous sections](#). Recall that the derivative of $f(x)$ was $f'(x)$ and the derivative of the $\sum \text{weights}$ was the respective input.

$$\begin{aligned} \Delta \text{weights}_{21} &= \text{error}_1 * f'(x) * \text{sum}'_{\text{weights}_{21}} = -1.8839 * 1 * -1.0193 = 1.9258 \\ \Delta \text{weights}_{22} &= \text{error}_1 * f'(x) * \text{sum}'_{\text{weights}_{22}} = -1.8839 * 1 * -0.4819 = 0.9105 \\ \Delta \text{weights}_{23} &= \text{error}_2 * f'(x) * \text{sum}'_{\text{weights}_{23}} = 0.4116 * 1 * -1.0193 = -0.4195 \\ \Delta \text{weights}_{24} &= \text{error}_2 * f'(x) * \text{sum}'_{\text{weights}_{24}} = 0.4116 * 1 * -0.4819 = -0.1984 \end{aligned}$$

In order to calculate deltas for the weights in the hidden layer, one more step needs to be considered. Since the weights of the hidden layer neurons affect not only the outcome of their own neuron, but also the outcome of neurons further 'downstream' to them (e.g. in this case the output neurons), their total influence needs to be taken into account when calculating the delta. In practise, this means that first, the backpropagation of the weight for output_1 needs to be calculated, and then the backpropagation of the weight for output_2 is calculated. Subsequently, these values are summed. In larger networks, with more hidden layers, these calculations can become pretty big and almost impossible to write down in formulas. That is why many machine learning libraries (such as Tensorflow²) do all of this math in the background. But now, for the network with a single hidden layer, the formula will be worked out fully.

Practical thinking about backpropagation

If it is conceptually hard to understand how the backpropagation might function in a larger network, it is best to draw (imaginary) lines from the current delta / weight you are trying to calculate to every possible output of the model. These lines now show which backpropagation calculations need to be performed and summed to get the final delta.

$$\begin{aligned} \Delta \text{weights}_{11} &= \text{error}_1 * f'(x) * \text{sum}'_{\text{weights}_{11}} * f'(x) * \text{sum}'_{\text{weights}_{21}} + \text{error}_2 * f'(x) * \text{sum}'_{\text{weights}_{23}} * f'(x) * \text{sum}'_{\text{weights}_{11}} \\ \Delta \text{weights}_{12} &= \text{error}_1 * f'(x) * \text{sum}'_{\text{weights}_{12}} * f'(x) * \text{sum}'_{\text{weights}_{21}} + \text{error}_2 * f'(x) * \text{sum}'_{\text{weights}_{23}} * f'(x) * \text{sum}'_{\text{weights}_{12}} \\ \Delta \text{weights}_{13} &= \text{error}_1 * f'(x) * \text{sum}'_{\text{weights}_{13}} * f'(x) * \text{sum}'_{\text{weights}_{21}} + \text{error}_2 * f'(x) * \text{sum}'_{\text{weights}_{23}} * f'(x) * \text{sum}'_{\text{weights}_{13}} \\ \Delta \text{weights}_{14} &= \text{error}_1 * f'(x) * \text{sum}'_{\text{weights}_{14}} * f'(x) * \text{sum}'_{\text{weights}_{22}} + \text{error}_2 * f'(x) * \text{sum}'_{\text{weights}_{24}} * f'(x) * \text{sum}'_{\text{weights}_{14}} \\ \Delta \text{weights}_{15} &= \text{error}_1 * f'(x) * \text{sum}'_{\text{weights}_{15}} * f'(x) * \text{sum}'_{\text{weights}_{22}} + \text{error}_2 * f'(x) * \text{sum}'_{\text{weights}_{24}} * f'(x) * \text{sum}'_{\text{weights}_{15}} \\ \Delta \text{weights}_{16} &= \text{error}_1 * f'(x) * \text{sum}'_{\text{weights}_{16}} * f'(x) * \text{sum}'_{\text{weights}_{22}} + \text{error}_2 * f'(x) * \text{sum}'_{\text{weights}_{24}} * f'(x) * \text{sum}'_{\text{weights}_{16}} \end{aligned}$$

$$\begin{aligned} \Delta \text{weights}_{11} &= -1.8839 * 1 * -1.0193 * 0.051 + 0.4116 * 1 * -1.0193 * 0.051 = 0.0768 \\ \Delta \text{weights}_{12} &= -1.8839 * 1 * -1.0193 * 0.467 + 0.4116 * 1 * -1.0193 * 0.467 = 0.7034 \\ \Delta \text{weights}_{13} &= -1.8839 * 1 * -1.0193 * 1.205 + 0.4116 * 1 * -1.0193 * 1.205 = 1.8150 \\ \Delta \text{weights}_{14} &= -1.8839 * 1 * -0.4819 * 0.051 + 0.4116 * 1 * -0.4819 * 0.051 = 0.0363 \\ \Delta \text{weights}_{15} &= -1.8839 * 1 * -0.4819 * 0.467 + 0.4116 * 1 * -0.4819 * 0.467 = 0.3326 \\ \Delta \text{weights}_{16} &= -1.8839 * 1 * -0.4819 * 1.205 + 0.4116 * 1 * -0.4819 * 1.205 = 0.8580 \end{aligned}$$

Now that all deltas are calculated, they need to be applied to the weights. As introduced in a previous section, it is often common to multiply the delta to a *learning rate*, in order to limit the influence of any single measurement (sample) on the model. In this

example, a learning rate of 0.01 will be applied. While this may seem small for this single calculation, keep in mind that these training calculations are often performed hundreds or even thousands of times.

$$\begin{aligned} \text{weight11}_{\text{new}} &= \text{weight11} - (\text{learning rate} * \Delta_{\text{weight11}}) = -0.50 - (0.01 * 0.0768) = -0.5008 \\ \text{weight12}_{\text{new}} &= \text{weight12} - (\text{learning rate} * \Delta_{\text{weight12}}) = 0.22 - (0.01 * 0.7034) = 0.2130 \\ \text{weight13}_{\text{new}} &= \text{weight13} - (\text{learning rate} * \Delta_{\text{weight13}}) = -0.91 - (0.01 * 1.8150) = -0.9282 \\ \text{weight14}_{\text{new}} &= \text{weight14} - (\text{learning rate} * \Delta_{\text{weight14}}) = 0.61 - (0.01 * 0.0363) = 0.6096 \\ \text{weight15}_{\text{new}} &= \text{weight15} - (\text{learning rate} * \Delta_{\text{weight15}}) = 0.14 - (0.01 * 0.3326) = 0.1367 \\ \text{weight16}_{\text{new}} &= \text{weight16} - (\text{learning rate} * \Delta_{\text{weight16}}) = -0.48 - (0.01 * 0.8580) = -0.4886 \\ \text{weight21}_{\text{new}} &= \text{weight21} - (\text{learning rate} * \Delta_{\text{weight21}}) = 0.73 - (0.01 * 1.9258) = 0.7107 \\ \text{weight22}_{\text{new}} &= \text{weight22} - (\text{learning rate} * \Delta_{\text{weight22}}) = 0.29 - (0.01 * 0.9105) = 0.2809 \\ \text{weight23}_{\text{new}} &= \text{weight23} - (\text{learning rate} * \Delta_{\text{weight23}}) = -0.29 - (0.01 * 0.4195) = -0.2858 \\ \text{weight24}_{\text{new}} &= \text{weight24} - (\text{learning rate} * \Delta_{\text{weight24}}) = -0.24 - (0.01 * 0.1984) = -0.238 \end{aligned}$$

Now that the weights have been tuned, check if the prediction error or loss has improved.

$$\begin{aligned} \text{hidden}_1 &= -1.0446 \\ \text{hidden}_2 &= -0.4938 \\ \text{output}_1 &= -0.8810 \\ \text{output}_2 &= 0.4161 \\ \text{error}_1_{\text{new}} &= \text{output}_1 - \text{expected output}_1 = -0.8810 - 1 = -1.8810 \\ \text{error}_2_{\text{new}} &= \text{output}_2 - \text{expected output}_2 = 0.4161 - 0 = 0.4161 \\ \text{loss}_{\text{new}} &= \text{error}_1^2 + \text{error}_2^2 = -1.4649 \end{aligned}$$

The initial error was (-1.4723) and the new error has reduced to (-1.4649) , meaning that the training was successful. Now the above steps need to be reproduced many times, and with different measurements, to ensure that the resulting model works for both gestures.

After manually working out one measurement training cycle, it may become apparent why it is good to let a computer program handle all the calculations. So, the training of the gesture recognition system will be put into a Python program.

To introduce some variance into the training, the measurements / samples are shuffled for each training cycle. This ensures that the model gets trained evenly and unbiased. The amount of training epochs and the learning rate were chosen after some experimentation. It is interesting to see what changes in the calculation if these are tweaked (when using machine learning libraries, it is also common to tune these parameters until the desired output or loss is achieved). See if the model is able to train successfully on the measurements recorded in the [last section](#) by adding your own measurements into the code.



training_model.py

```
# Import the random library to initialise the weights
import random

### Put the recorded measurements here ###
moving_closer = [
    [0.0, 35.5, 120.1],
    [2.625, 33.375, 127.875],
    [19.0, 56.375, 127.875],
    [47.875, 51.375, 127.875],
    [40.125, 89.375, 127.875]
]

moving_away = [
    [104.54, 2.375, 0.0],
    [127.875, 33.625, 5.4],
    [120.3, 39.125, 0.0],
    [127.875, 103.25, 23.8],
    [127.875, 29.25, 0.0]
]

# Labeling the collected data, outputting the probability of that gesture
# output1 = p(moving closer), output2 = p(moving away)
moving_closer_labeled = [(sample, [1, 0]) for sample in moving_closer]
moving_away_labeled = [(sample, [0, 1]) for sample in moving_away]

# Making one pile of samples from the split samples above
samples = moving_closer_labeled + moving_away_labeled

# Initialise the weights for the network randomly, define the learning rate and epochs (iterations)
weights = [
    [random.uniform(-0.1, 0.1) for _ in range(6)],
```

```

[random.uniform(-0.1, 0.1) for _ in range(4)]
]
learning_rate = 0.01
epochs = 100

# Define the activation function (linear)
def activation(x):
    return x

# Run the training for the configured epochs
for epoch in range(epochs):

    # Shuffle the samples in order to introduces some variance into the training
    random.shuffle(samples)

    # Define a total loss for the whole system
    loss = 0.0

    # Run through all samples
    for sample in samples:

        # Separate the inputs and expected outputs from the sample, pre-processing the inputs
        inputs = [s/100 for s in sample[0]]
        expected = sample[1]

        # Calculate the output of the hidden layer
        hiddens = [
            activation(inputs[0] * weights[0][0] + inputs[1] * weights[0][1] + inputs[2] * weights[0][2]),
            activation(inputs[0] * weights[0][3] + inputs[1] * weights[0][4] + inputs[2] * weights[0][5])
        ]

        # Calculate the output from the output layer
        outputs = [
            activation(hiddens[0] * weights[1][0] + hiddens[1] * weights[1][1]),
            activation(hiddens[0] * weights[1][2] + hiddens[1] * weights[1][3]),
        ]

        # Calculate the errors
        errors = [
            outputs[0] - expected[0],
            outputs[1] - expected[1]
        ]

        # Add the errors to the system loss
        loss += errors[0] + errors[1]

        # Calculate the deltas for each weight
        deltas = [
            [
                errors[0] * 1 * hiddens[0] * 1 * inputs[0] + errors[1] * 1 * hiddens[0] * 1 * inputs[0],
                errors[0] * 1 * hiddens[0] * 1 * inputs[1] + errors[1] * 1 * hiddens[0] * 1 * inputs[1],
                errors[0] * 1 * hiddens[0] * 1 * inputs[2] + errors[1] * 1 * hiddens[0] * 1 * inputs[2],
                errors[0] * 1 * hiddens[1] * 1 * inputs[0] + errors[1] * 1 * hiddens[1] * 1 * inputs[0],
                errors[0] * 1 * hiddens[1] * 1 * inputs[1] + errors[1] * 1 * hiddens[1] * 1 * inputs[1],
                errors[0] * 1 * hiddens[1] * 1 * inputs[2] + errors[1] * 1 * hiddens[1] * 1 * inputs[2],
            ],
            [
                errors[0] * 1 * hiddens[0],
                errors[0] * 1 * hiddens[1],
                errors[1] * 1 * hiddens[0],
                errors[1] * 1 * hiddens[1]
            ]
        ]

        # Apply the calculated deltas to the weights, keeping in mind the learning rate
        for layer in range(len(weights)):
            for weight in range(len(weights[layer])):
                weights[layer][weight] -= learning_rate * deltas[layer][weight]

    # Print the loss every 10 epochs in order to check progress
    if epoch % 10 == 0:
        print(f"epoch:{epoch} {loss=}")

# Print the final loss of the system and the weights
print(f"loss={loss}")
print(f"weights={weights}")

```

Training problems

Training a neural network needs insights and sometimes even luck. Below are some points of attention to look at when the training does not work like intended.

1. One or more of the weights of the model have exploded (gone far into the negative or positive), subsequently skewing the whole model. This can be solved by reducing the learning rate, or by limiting the weights.
2. One or more of the samples used in training are skewed or an outlier. This can lead to the model not being able to reach acceptably tuned weights. This can be solved by excluding edge cases, or by changing the network structure (shape).
3. The model outputs `NaN` (Not a Number), if the model is not able to output the weights anymore, this usually means that they have exploded (see point 1 above).
4. The model does not seem to reach an acceptable loss, even after hundreds or thousands of epochs. This could be caused because a network structure (shape) was chosen that does not fit well with the data it tries to predict.

If the loss at the end of training is satisfactory (e.g. it has decreased a lot), the weights seem to be tuned well. Now they can be loaded into a model that runs on the TinySpark development kit, predicting gestures.

In the next section, the model will be transferred to the TinySpark development kit, and the trained weights will be loaded, in order to test the trained model.

1. https://en.wikipedia.org/wiki/Loss_function ↩
2. <https://www.tensorflow.org/> ↩

5.5 Gesture recognition - deploying the model

Now it is time to deploy the trained model to the TinySpark development board. In order to do so, the recording code from [a previous section](#) is rewritten to accommodate the prediction. The trained weights are included in the code. Since the training phase of the neural network is not needed anymore, its code will not be included from [the previous section](#). The program will work by pressing **Button 1** and then recording a gesture. The on-board LED will be on when recording the gesture. The final prediction (the gesture with the highest probability) is printed to the serial console together with its probability.

 [Open code](#)

gesture_model.py

```
# Import all libraries
import time
import board
from apds9930.apds9930 import APDS9930
from digitalio import DigitalInOut, Direction, Pull

# Initialize I2C
i2c = board.I2C()
sensor = APDS9930(i2c)

# Initialise buttons 1 and 2
button1 = DigitalInOut(board.BUTTON1)
button1.direction = Direction.INPUT
button1.pull = Pull.UP
button2 = DigitalInOut(board.BUTTON2)
button2.direction = Direction.INPUT
button2.pull = Pull.UP

# Initialise LED
led = DigitalInOut(board.LED)
led.direction = Direction.OUTPUT

### Put the trained weights here ###
weights=[[0.664838063268332, 0.14475863419970098, 0.39798479496743655, -0.1996792640967368, -0.10042784907697525, -0.38665016231765525], [0.19784347834163832,
-0.7106415457739413, 0.6545148412015216, 0.1258938146460059]]

# Define the activation function (linear)
def activation(x):
    return x

# Loop continuously
while True:

    # Check button 1 (note the signal is pulled up / high, so check for low signal)
    if button1.value == 0:

        # Startup delay
        recording = list()
        print("> Starting recording in 1sec")
        time.sleep(1)

        # Start recording
        led.value = 1
        distance = sensor.proximity
        recording.append(distance)
        print(f"distance={}")
        # delay for 250ms
        time.sleep(0.250)
        distance = sensor.proximity
        recording.append(distance)
        print(f"distance={}")
        # delay for another 250ms
        time.sleep(0.250)
        distance = sensor.proximity
        recording.append(distance)
        print(f"distance={}")

        # Stop recording
        led.value = 0
        print("> Stopped recording")

        # Start predicting
        inputs = [r/100 for r in recording]

        # Calculate the output of the hidden layer
        hiddens = [
            activation(inputs[0] * weights[0][0] + inputs[1] * weights[0][1] + inputs[2] * weights[0][2]),
            activation(inputs[0] * weights[0][3] + inputs[1] * weights[0][4] + inputs[2] * weights[0][5])
        ]

        # Calculate the output from the output layer
        outputs = [
```

```

        activation(hiddens[0] * weights[1][0] + hiddens[1] * weights[1][1]),
        activation(hiddens[0] * weights[1][2] + hiddens[1] * weights[1][3]),
    ]

    # Print the prediction, keeping in mind that the output with the highest probability is picked
    if outputs[0] >= outputs[1]:
        print(f"Prediction: moving closer, p={outputs[0]}")
    else:
        print(f"Prediction: moving away, p={outputs[1]}")

    # Debounce the buttons
    time.sleep(0.01)

```

Deploy the model to the TinySpark development kit and add your trained weights to it. Press **Button 1** and see if the model predicted the gesture correctly.

Deployment problems

The trained model might behave differently than expected once it has been uploaded to the TinySpark development board. This could be due to a plethora of reasons, some are listed below.

1. The measurement taken was not performed in the same way as the recordings during training (e.g. using a piece of cardboard vs your hand).
2. The measurement taken was not performed in the same lighting conditions as the recordings during training (e.g. inside a dimly lit room vs outside).
3. The model was trained using a narrow range of measurements (all measurements resembled each other very closely), this can lead to a skewed prediction model.
4. The model was trained with too little samples.

In the next chapter, there will be some closing remarks, recommendations for further learning and project ideas to get you started on your own TinyML journey.

6. Beyond TinySpark

6.1 Dive into TinyML

This chapter will be used to give some closing remarks, recommendations for further learning and project ideas to get started on your own TinyML powered projects.

6.1.1 Closing remarks

I hope that through the TinySpark platform and development kit, you have been able to lift the 'black box' that is often associated with machine learning (and subsequently TinyML). Through the step-by-step explanation of concepts and the mini-projects, you should have gained a better understanding of the mathematics and logic behind (Tiny) machine learning, as well as its application areas. While there are certainly many more nuances to be learned regarding machine learning, and deployment to TinyML capable devices, you should now have a solid basis to start from.

6.1.2 Continue learning TinyML

As could already be seen in the [previous chapter](#), there are quite some calculations involved with the training and deployment of neural networks. As there is much mathematics involved, it can quickly become quite overwhelming and opaque again. If you plan on further developing more extensive neural networks using the techniques learned here, it is recommended to write a Python library or a class for handling all of the bookkeeping and calculations. Good starting points include the tutorial by [Déborah Mesquita on Real Python](#), the tutorial by [Dr. Michael J. Garbade on KDnuggets](#) and the video series by [Andrej Karpathy called Zero to AI Hero](#).

Alternatively, open source libraries such as [Tensorflow](#) and [PyTorch](#) can be used to setup and train neural networks. These Python libraries support many more advanced features, and are maintained and updated regularly. For deploying models trained by Tensorflow, you can use the [Tensorflow Lite Micro](#) framework, that is very easily integrated into normal Tensorflow code. Specific examples that can be used on the TinySpark development kit can be found on the [Espressif TFLite Micro examples page](#). *Be aware that these machine learning frameworks produce neural networks that are not easily viewed or edited, as they are often deployed as compressed C code.*

To learn more about TinyML in general, the [TinyML foundation](#) hosts regular (online) talks with industry experts. Harvard University heads the [Tiny Machine Learning Open Education Initiative](#) which includes many resources from (free) courses, online tutorials, code repositories and much more. Lastly, the [MIT HAN lab](#) has interesting examples and explanations on the compression and acceleration of neural networks and models.

6.1.3 Project ideas

If you want to continue and develop your own TinyML powered projects using the TinySpark development board, here are some project ideas:

- Vibration detection by analysing time series data from the Inertial Motion sensor
- Wake word detection using the on-board Microphone
- Weather change prediction using the Pressure sensor
- Game intelligence such as Tic-Tac-Toe or Snake
- Package handling analysis using the data from the Inertial Motion sensor
- Sentiment analysis from the pitch of voices using the Microphone
- Advanced plant monitoring using the Environmental sensor and Light sensor
- Infrared decoding of transmission signals using the on-board IR receiver
- Morse code decoding by time interval measurement
- Food quality prediction using the Environmental sensor
- Fall detection by analysing impact data from the Inertial Motion sensor
- Motor malfunction detection using the Hall effect sensor

By connecting some external sensors using the Expansion header or Stemma QT / Qwiic connector, some more interesting projects could be made:

- Perfect al-dente pasta cooking through temperature measurement
- Pulse monitoring and fatigue detection using a pulse-oximeter
- Leak detection using flow sensing valves
- Navigational aid using LIDAR sensing
- Fruit ripeness detection by using a colour sensor
- Soil condition determination by measuring moisture and acidity

6.1.4 Featured projects

If you have finished an interesting project and want it displayed here, please contact me via j-siderius@GitHub.

-

7. About

7.1 TinySpark

The TinySpark project is created by J Siderius, as part of the Bachelor Thesis for Creative Technology in 2023.

The projects' aim is to enable everyone, from hobbyist, maker, student to professor, to understand (Tiny) Machine Learning in a simple and intuitive way. Furthermore, by providing many sample projects, users should be able to integrate meaningful TinyML applications into their own projects and needs.

7.2 TinySpark licenses

All content of the TinySpark project is licensed under [CC BY-SA 4.0](#), the underlying source code is licensed under [GNU AGPLv3](#).

If you are interested in using (parts of) the contents on this platform, feel free to contact me via j-siderius@GitHub.

Visit the original repository for more information: [TinySpark/j-siderius@Github](#).

© 2023 by JC Siderius