

AI REGELSYSTEEM

PROJECTDOCUMENT



Projectleden

Thomas van Ommeren

Yered Scheffer

Mitchel Reints

Piet Poot

Begeleider

Daniel Versluis

Opdrachtgever

HR Datalab EAS

Stefan Groot Nibbelink

Datum van uitgifte

22-06-2025

INHOUDSOPGAVE

Versiehistorie.....	4
1: Inleiding	5
1.1: Doel	5
1.2: Documentconventies	5
1.3: Leeswijze	5
2: Eisen en acceptatiecriteria.....	5
2.1 Functionele eisen	6
2.1.1 Doel van het systeem.....	6
2.1.2 Regelfunctie	6
2.1.3 AI-functionaliteit	6
2.1.4 Inputvariabelen voor het AI-model.....	6
2.2 Dataset- en trainingseisen.....	6
2.2.1 Dataset-eisen	6
2.2.2 Modelarchitectuur	7
2.3 Hardware-eisen	7
2.3.1 Sensoren en actuatoren.....	7
2.3.2 Microcontroller	7
2.4 Software-eisen	7
2.4.1 Ontwikkelomgeving en code.....	7
2.4.2 Logging en monitoring	7
3: Scrum aanpak	8
4: Sprints.....	8
4.1: Sprint 0 & 1 – Voorbereiding en opstart	9
4.2: Sprint 2 – Eisen en documentatie	9
4.2.2: Simple AI	10
4.2.3: hardware implementatie	13
4.2.4: LSTM en Complex AI	15
4.2.5: binnen VScode testen	18
4.3: Sprint 3 – AI-modellen en simulatie	24
4.3.1 Simulatie	24
4.4: Sprint 4 – Dataset uitbreiden,verfijnen en importeren model op de microcontroller.....	27
4.4.1 Importeren model op de microcontroller.....	27
4.5: Sprint 5 –Zelf ontwikkeldmodel en integratie	35
4.5.1: Zelfontwikkelde model	35
4.5.2: Eiq model	42
4.6: Sprint 6 – Afronding en oplevering	42
5: onderzoek.....	43

5.1: NXP Toolkit (eiQ)	43
5.2: onderzoek mitchel en thomas (Is het mogelijk om een PID controller te vervangen door een AI?)	43
6: implementatie	44
6.1: zelfontwikkelde model	44
6.2: EIQ model.....	44
6.2.1: Dataset en modelselectie in de eiQ Toolkit	44
6.2.2: Modelselectie: regressiemodellen.....	44
6.2.3: Evaluatie van de modellen.....	45
6.2.4: intergratie	46
7: Conclusie.....	48
8: Bibliografie.....	48
9: Bijlagen	49
9.1: Bijlage 1 – Onderzoeksverslag – mogelijkheden ontwikkelen AI met NXP tools.....	49
INLEIDING	49
BASISWERKING SOFTWARE.....	49
GENEREREN DATASET	49
MODEL TRAINING	50
<i>Projectconfiguratie</i>	50
<i>Datasetbeheer</i>	50
<i>Modeltraining</i>	51
MODEL OPTIMALISATIE	51
<i>Balans van de data</i>	51
<i>Samenhang tussen kanalen:</i>	51
<i>Belang van de kanalen</i>	51
<i>Advies voor de meetinstellingen</i>	51
IMPLEMENTATIE	52
CONCLUSIE.....	52
9.2: Bijlage 2 – Onderzoeksverslag – Is het mogelijk om een PID controller te vervangen Door een AI?	53
uitslagen Complexe AI's.....	60
eerste toepassing	60

VERSIEHISTORIE

Versie	Datum	Wijzingen	Auteur
0.1	22-5-2025	Eerste opzet eind-projectdocument	Piet
0.2	5-6-2025	Opzet onderzoek en eisen	Piet
0.3	18-6-2025	Uitwerking sprints en scrum aanpak	Piet
0.4	19-6-2025	Sprints documentatie	Yered En Piet
0.5	19-6-2025	Toevoeging LSTM	Yered
1	22-6-2025	Volledige versie	Piet, Yered, Thomas en Mitchel

1: INLEIDING

Dit hoofdstuk vormt de inleiding van dit projectdocument en beschrijft het doel van het project, de gebruikte documentconventies en de opbouw van het document.

1.1: DOEL

Het systeem dat in dit project ontwikkeld wordt, is een AI-gebaseerd regelsysteem voor de ELE ping-pong-installatie. Deze installatie houdt met behulp van een ventilator een pingpongbal op een gewenste hoogte in een transparante buis. De installatie dient als testomgeving voor het toepassen van moderne regeltechnieken op embedded systemen.

Het doel van dit systeem is om te onderzoeken hoe kunstmatige intelligentie ingezet kan worden voor regeltoepassingen op microcontrollers. Waar traditionele regeltechniek (zoals PID-regelaars) beperkingen kent bij veranderende of complexe omstandigheden, kan AI zichzelf aanpassen en zonder vooraf gedefinieerd model functioneren. De AI-oplossing moet stabiel zijn, flexibel omgaan met omgevingsveranderingen, en volledig autonoom werken — dus zonder constante internetverbinding of afhankelijkheid van externe servers.

Naast het technische onderzoek fungeert het systeem als demonstratiemodel voor onderwijsdoeleinden. Het wordt ingezet tijdens open dagen, presentaties en onderwijsactiviteiten binnen de opleiding Elektrotechniek en het HR Datalab EAS. Hiermee wordt AI-regeltechniek tastbaar gemaakt voor studenten en wordt ervaring opgebouwd die kan worden ingezet in toekomstige projecten.

1.2: DOCUMENTCONVENTIES

In dit document zijn de volgende conventies gehanteerd:

- Koppen zijn genummerd volgens het decimale systeem (bijvoorbeeld 1.1, 1.2).
- Belangrijke termen worden cursief weergegeven en toegelicht bij de eerste introductie.
- Alle tabellen en figuren zijn genummerd en voorzien van een bijschrift.
- Bronvermeldingen volgen de IEEE-referentiestijl.

1.3: LEESWIJZE

Dit document is chronologisch opgebouwd, zodat de voortgang van het project stap voor stap gevuld kan worden. Voor een volledig beeld wordt aanbevolen het document van begin tot eind te lezen. Voor een beknopt overzicht van de resultaten en inzichten kunt u ook direct naar hoofdstuk 8: Conclusie gaan.

2: EISEN EN ACCEPTATIECRITERIA

Voorafgaand aan de start van het project zijn er specifieke eisen en acceptatiecriteria opgesteld om richting te geven aan de ontwikkeling van het pingpong AI-systeem. Deze criteria vormen de basis waarop beslissingen worden genomen tijdens het ontwikkelproces en bepalen of het eindresultaat voldoet aan de vooraf gestelde verwachtingen. Ze zorgen ervoor dat zowel de functionele werking als de prestaties van het systeem objectief beoordeeld kunnen worden. In de onderstaande koppen worden deze eisen beschreven.

In dit hoofdstuk worden de eisen beschreven waaraan het systeem moet voldoen. De eisen zijn onderverdeeld in functionele eisen, dataset- en trainingseisen, hardware-eisen en software-eisen.

2.1 FUNCTIELE EISEN

2.1.1 DOEL VAN HET SYSTEEM

- Het systeem moet een object (pingpongbal) zwevend houden op een ingestelde hoogte.

2.1.2 REGELFUNCTIE

- Het systeem moet de hoogte van het object regelen met een afwijking van maximaal 5 cm na 30 seconden.
- De gewenste hoogte is instelbaar tussen 750 mm en 900 mm.
- Het object mag de opstelling niet verlaten tijdens de werking.

2.1.3 AI-FUNCTIONALITEIT

- De regeling wordt uitgevoerd door een neuraal netwerk.
- Het netwerk moet het gedrag van een klassieke PID-regelaar kunnen benaderen.
- Het netwerk genereert een PWM-singaal (0–100%) voor de aansturing van de ventilator.
- Het netwerk moet in real-time werken en periodiek nieuwe outputwaarden berekenen.
- Het netwerk moet vergelijkbare prestaties leveren als een PID-regelaar binnen een gesloten regelsysteem.

2.1.4 INPUTVARIABELEN VOOR HET AI-MODEL

- Het neuraal netwerk gebruikt minimaal de volgende inputs:
 - Gemeten hoogte (huidige waarde)
 - Gewenste hoogte (setpoint)
 - Huidige PWM-waarde

2.2 DATASET- EN TRAININGSEISEN

2.2.1 DATASET-EISEN

- Het neuraal netwerk wordt getraind op basis van data uit:
 - Gesimuleerde datasets
 - Metingen van de bestaande PID-opstelling.
- Tijdens de training worden verschillende scenario's toegepast, waaronder:
 - Plotselinge veranderingen
 - Trapsgewijze referenties
 - Toegevoegde verstoringen (ruis)
- De AI mag pas geïmplementeerd worden zodra een nauwkeurigheid van minimaal 80% is behaald tijdens de validatie.

2.2.2 MODELARCHITECTUUR

- Het netwerk bevat minimaal drie lagen: inputlaag, minimaal één verborgen laag (hidden layer) en een outputlaag.
- De architectuur moet lichtgewicht zijn, met een beperkt aantal parameters, zodat het geschikt is voor embedded hardware.

2.3 HARDWARE-EISEN

2.3.1 SENSOREN EN ACTUATOREN

- De hoogte van het object wordt gemeten met een ultrasoensor.
- De gemeten hoogte wordt weergegeven in millimeters.
- De ventilator wordt aangestuurd via PWM (0–100%).

2.3.2 MICROCONTROLLER

- Het systeem draait volledig op één microcontroller, bij voorkeur de FRDM-MCXN947.
- De AI moet functioneren op een microcontroller met beperkte rekenkracht en geheugen

2.4 SOFTWARE-EISEN

2.4.1 ONTWIKKELOMGEVING EN CODE

- Het neurale netwerk wordt getraind in Python.
- De implementatie op de microcontroller wordt uitgevoerd in C.
- Het model moet exporteerbaar zijn naar C/C++ of via bibliotheken zoals TensorFlow Lite.
- De regeling draait volledig lokaal op de microcontroller, zonder afhankelijkheid van externe systemen.

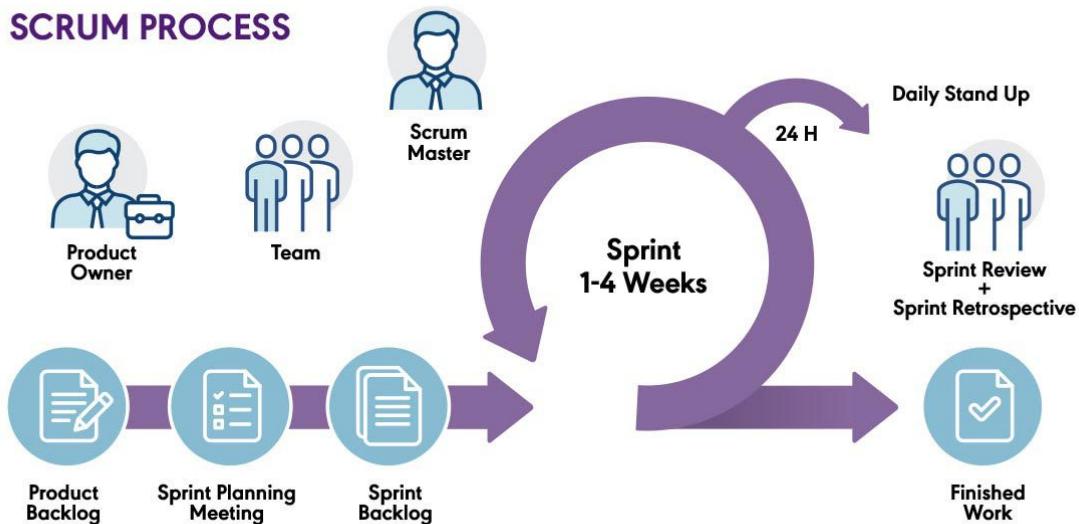
2.4.2 LOGGING EN MONITORING

- Tijdens de testfase moeten alle input- en outputwaarden van het netwerk worden gelogd.

3: SCRUM AANPAK

Voor de uitvoering van dit project is gekozen voor de Scrum-methodiek. Scrum is een agile werkwijze die zich kenmerkt door het werken in korte, herhalende cycli, zogenaamde *sprints*. Binnen elke sprint worden duidelijke doelen gesteld en wordt toegewerkt naar een werkend (tussen)resultaat. Deze aanpak maakt het mogelijk om flexibel in te spelen op nieuwe inzichten of onverwachte uitdagingen gedurende het project.

SCRUM PROCESS



Figuur 1: Scrum principe [1]

Aan het begin van het project is de globale planning opgesteld en is het werk verdeeld in overzichtelijke taken. Deze taken zijn vastgelegd in het projectmanagementsysteem [Notion](#), waarmee het team continu inzicht had in de voortgang.

Elke sprint begon met een korte planningssessie, waarin werd besproken welke taken in die sprint opgepakt zouden worden. Tijdens de uitvoering van de sprint werd regelmatig kort afgestemd over de voortgang en eventuele knelpunten. Aan het einde van iedere sprint vond een evaluatie plaats, waarin de behaalde resultaten werden besproken en waar nodig de planning voor de volgende sprint werd bijgesteld.

Dankzij deze iteratieve werkwijze konden verschillende onderdelen van het project parallel worden ontwikkeld, getest en verbeterd. Dit was met name waardevol, omdat zowel de hardware-implementatie als de ontwikkeling van het neurale netwerk in meerdere fasen tot stand kwamen.

4: SPRINTS

Het project bestond grofweg uit drie belangrijke onderdelen, die gezamenlijk hebben geleid tot de gekozen werkwijze en indeling in sprints. Allereerst was het van belang om een stabiele hardware-opstelling te realiseren waarbij de FRDM-MCXN947 microcontroller de pingpongbal op een vaste hoogte kan houden. Dit vroeg om een juiste aansturing van de ventilator en een nauwkeurige terugkoppeling vanuit de sensoren.

Vervolgens lag de focus op het ontwikkelen en trainen van een neurale netwerk dat in staat is het gedrag van een PID-regelaar na te bootsen. Dit netwerk moest op basis van de sensorwaarden voorspellen welke aansturing nodig is om de bal stabiel te houden.

Als derde onderdeel moest het getrainde neurale netwerk worden geïmplementeerd op de FRDM-MCXN947 microcontroller, zodat het model in real-time kan draaien en zelfstandig de regeling kan uitvoeren.

Deze drie onderdelen liepen gedeeltelijk parallel en vormden de basis voor de opzet van de verschillende sprints die hieronder worden toegelicht.

4.1: SPRINT 0 & 1 – VOORBEREIDING EN OPSTART

Het project is gestart met het opstellen van een projectplan en een plan van aanpak. Hierin zijn de doelstellingen, fasering en werkwijze vastgelegd. Er is gekozen om volgens de Scrum-methodiek te werken, zodat we iteratief konden ontwikkelen in korte sprints met tussentijdse evaluatiemomenten.

In deze fase is de taakverdeling bepaald en zijn de werkzaamheden en deadlines vastgelegd in Notion, zodat er vanaf het begin structuur en overzicht was. Er is bewust gekozen om het project op te splitsen in twee hoofdlijnen:

1. Het werken met de hardware-opstelling.
2. Het ontwikkelen en trainen van een AI-model.

Op advies van de begeleiders is al in een vroeg stadium gestart met de hardware: de FRDM-MCXN947 microcontroller. Er is begonnen met het nabouwen van de pingpongbal-opstelling uit de DIS10-cursus. Hierbij lag de focus op het realiseren van een stabiele regeling waarbij de bal op een vaste hoogte blijft zweven. Tegelijkertijd zijn we gestart met de eerste experimenten met AI-modellen, waarin met Python, TensorFlow en PyTorch werd gewerkt om inzicht te krijgen in de basisprincipes van neurale netwerken en trainingsprocessen.

4.2: SPRINT 2 – EISEN EN DOCUMENTATIE

In Sprint 2 zijn de functionele en technische eisen uitgewerkt. Hierbij is onder andere vastgelegd wat de output van het AI-model moest zijn, welke data nodig was en welke prestaties verwacht werden. De eerste ervaringen met de hardware en dataverzameling zijn hierbij ook gedocumenteerd. Deze documentatie vormde een belangrijke basis voor de vervolgstappen.

Daarnaast is er een dataset gegenereerd vanuit de pingpongopstelling met de PID-regelaar, zoals hieronder weergegeven.



Figuur 2: Eerste gegenereerde dataset

Tijdens deze sprint is het team opgesplitst in twee subgroepen om efficiënter te werken aan verschillende onderdelen van het project:

- Yered en Mitchel richtten zich op het verdiepen in kunstmatige intelligentie (AI) met het oog op onze toepassing. Hun doel was om meer inzicht te krijgen in hoe AI werkt en zelf getrainde modellen op te zetten. Binnen deze subgroep lag de focus op twee onderdelen:
 - Mitchel werkte aan het opzetten eenvoudige AI-modellen om basisprincipes te begrijpen.
 - Yered richtte zich op complexere AI-modellen. Hij trainde verschillende netwerken op de PID-dataset en paste deze vervolgens toe op de simulatie-dataset om prestaties te analyseren.
- Thomas en Piet onderzochten de mogelijkheden voor implementatie en gebruik van de development boards die we voor dit project ontvangen hebben. Daarnaast begonnen zij met het verkennen van methoden om zelf een geschikte dataset te genereren voor het trainen en testen van de AI.

4.2.2: SIMPLE AI

In deze sprint zijn we bezig geweest met doorlopen van een aantal simpele voorbeeld neurale netwerken. Het eerste voorbeeld van een neurale netwerk waarmee je cijfers kan herkennen. Dit voorbeeld hielp ons om inzicht te krijgen in de verschillende stappen van het trainingsproces, zoals het voorbereiden van de data, het definiëren van de modelarchitectuur en het optimaliseren. Dit voorbeeld maakte gebruik van de MNIST dataset, een veelgebruikte dataset voor het trainen van neurale netwerken. Deze dataset bevat afbeeldingen van handgeschreven cijfers en is ideaal voor het leren van de basisprincipes van beeldherkenning met neurale netwerken. Door dit voorbeeld te bestuderen, konden we de concepten van input- en outputlagen, activatiefuncties en backpropagation beter begrijpen. Dit vormde een solide basis voor de latere ontwikkeling. Een groot nadeel van dit voorbeeld was dat jezelf de formule voor bijvoorbeeld de acticatiefunctie moest implementeren. Dit was niet handig omdat dit voor een beginner zeer fout gevoelig is en hierdoor niet veel succes mee geboekt. Daarom hebben we besloten om de volgende voorbeelden te maken met TensorFlow en PyTorch. Deze frameworks bieden een veel gebruiksvriendelijker interface voor het bouwen en trainen van neurale netwerken, waardoor we ons konden richten op het invoeren van data en het instellen van parameters, in plaats van het handmatig implementeren van alle formules.

Daarna zijn we verder gegaan met tensorflow voorbeelden omdat dit ook goed aansloot bij de microcontroller want TensorFlow Lite is speciaal ontworpen voor embedded systemen. We hebben met tensorflow ook een voorbeeld gebruikt waarmee je geschreven cijfers kan herkennen. Dit was vooral handig omdat we nu ook konden zien hoe je een neurale netwerk kan trainen en testen met dit specifieke framework. Voor dit voorbeeld was meer succes geboekt met het trainen van het neurale netwerk, waardoor we een beter begrip kregen van de werking van neurale netwerken en hoe deze kunnen worden toegepast in dit specifieke framework.

Voor dit voorbeeld hebben we een dataset gebruikt die bestaat uit afbeeldingen van handgeschreven cijfers, waarbij elk cijfer is gelabeld met het juiste getal. Het doel was om een neurale netwerk te trainen dat in staat is om deze cijfers correct te classificeren op basis van de afbeeldingen. We hebben verschillende architecturen en hyperparameters getest om de nauwkeurigheid van het model te optimaliseren. De dataset komt van de MNIST-dataset, die veel wordt gebruikt voor het trainen van neurale netwerken. Deze dataset bevat 60.000 trainingsvoorbeelden en 10.000 testvoorbeelden van handgeschreven cijfers, waardoor het een uitstekende basis vormt voor het leren van de basisprincipes van beeldherkenning met neurale netwerken.

Eerste moeten we de packages installeren die we nodig hebben voor het trainen van het neurale netwerk. We hebben hier voor de volgende commando's gebruikt:

```
pip install tensorflow matplotlib numpy
```

vervolgens moeten we de packages aanroepen in de code:

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
```

daarna moeten we de dataset inladen en voorbereiden. We hebben hiervoor de volgende code gebruikt:

```
# Laad de MNIST dataset
mnist = tf.keras.datasets.mnist
(training_data, training_labels), (test_data, test_labels) = mnist.load_data()

# Normaliseer de pixelwaarden
training_data = training_data / 255.0
test_data = test_data / 255.0
```

Nu hebben we de data ingeladen en genormaliseerd, zodat de pixelwaarden tussen 0 en 1 liggen. Dit is belangrijk voor het trainen van het neurale netwerk, omdat het helpt om de convergentie te versnellen en de prestaties te verbeteren. Nu kunnen we het neurale netwerk definiëren. We hebben hiervoor de volgende code gebruikt:

```
# Definieer het neurale netwerk
model = tf.keras.Sequential([
    tf.keras.Input(shape=(28, 28)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy']) # Inputlaag
```

In deze code hebben we een eenvoudig neurale netwerk gedefinieerd met een inputlaag, een verborgen laag en een outputlaag. De inputlaag heeft een vorm van (28, 28), wat overeenkomt met de afmetingen van de afbeeldingen in de MNIST-dataset. De verborgen laag heeft 128 neuronen en gebruikt de ReLU-activatiefunctie, terwijl de outputlaag 10 neuronen heeft en de softmax-activatiefunctie gebruikt om de waarschijnlijkheid van elk cijfer te berekenen. We hebben ook de optimizer en verliesfunctie gedefinieerd die we gaan gebruiken voor het trainen van het model. Nu kunnen we het neurale netwerk trainen met de volgende code:

```
# Train het neurale netwerk
model.fit(training_data, training_labels, epochs=5)
```

In deze code hebben we het neurale netwerk getraind met de trainingsdata en -labels voor 5 epochs. Dit betekent dat het model de trainingsdata 5 keer doorloopt en zijn gewichten bijwerkt op basis van de fout die het maakt bij het voorspellen van de labels. Na het trainen van het model kunnen we de prestaties evalueren met de volgende code:

```
# Evalueer het neurale netwerk
model.evaluate(test_data, test_labels)
```

In deze code hebben we het neurale netwerk geëvalueerd met de testdata en -labels. Dit geeft ons een idee van hoe goed het model presteert op ongeziene data. We verwachten een nauwkeurigheid van ongeveer 97% of hoger, wat aangeeft dat het model in staat is om de cijfers correct te classificeren. We kunnen ook enkele voorspellingen doen met het getrainde model en de resultaten visualiseren.

```

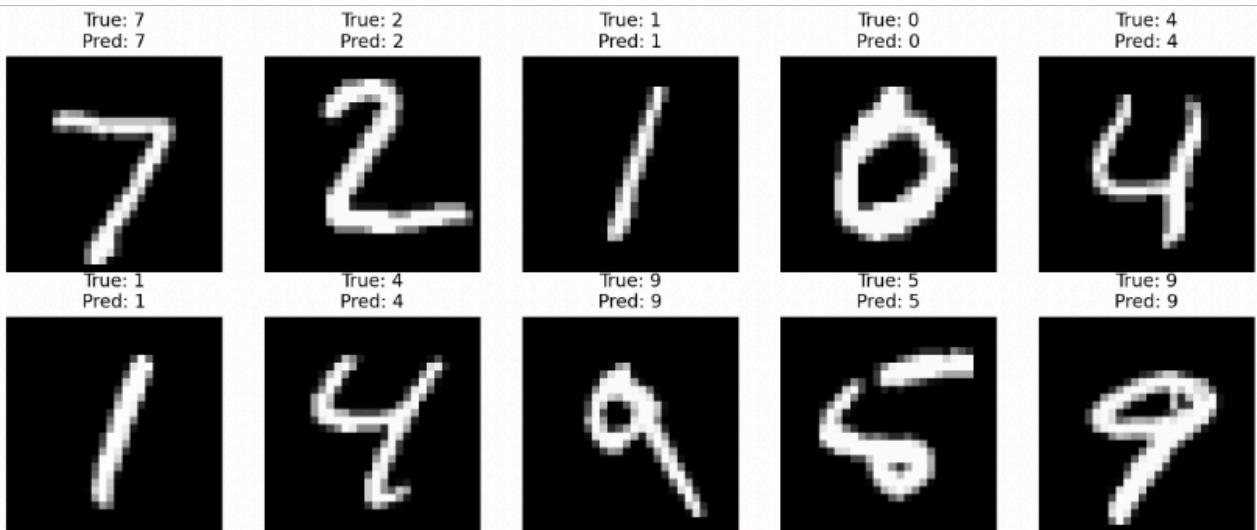
# Voorspellingen
predictions = model.predict(test_data)
np.set_printoptions(suppress=True)
print("Voorbeeld testlabel:", test_labels[1])
print("Voorspelling:", predictions[1])

# Plot voorbeelden
def plot_examples(test_data, test_labels, predictions, num_examples=10):
    plt.figure(figsize=(12, 5))
    for i in range(num_examples):
        plt.subplot(2, 5, i + 1)
        plt.imshow(test_data[i], cmap='gray')
        plt.title(f"True: {test_labels[i]}\nPred: {np.argmax(predictions[i])}")
        plt.axis('off')
    plt.tight_layout()
    plt.show()

plot_examples(test_data, test_labels, predictions)

```

In deze code hebben we enkele voorspellingen gedaan met het getrainde model en de resultaten gevisualiseerd. Zie foto hieronder voor een voorbeeld van de uitkomst van dit neuraal netwerk.



4.2.3: HARDWARE IMPLEMENTATIE

Om de AI werkend te krijgen op de microcontroller, moet eerst de oorspronkelijke PID-regeling functioneren op de microcontroller. Aangezien de eerdere PID al functioneel was op een Texas Instruments-microcontroller, moest de code deels aangepast en bewerkt worden om deze ook werkend te krijgen op het nieuwe NXP-bord.

De aansturing van de ventilator gebeurt via een softwarematige PWM-generatie met behulp van een hardwaretimer (CTIMER) op de microcontroller.

Hiervoor wordt een interne teller (timer0) gebruikt, die bij elke timer-interrupt met 1 wordt verhoogd. Zodra timer0 een vooraf ingestelde bovengrens (TIMERTOP0) bereikt, wordt de teller weer op nul gezet. Op deze manier ontstaat een repeterende telcyclus met vaste frequentie, die dient als tijdsbasis voor het PWM-signal.

```
82     if(timer0 >= TIMERTOP0)
83     {
84         timer0 = 0;
85         timer0Expired = true;
86     }
```

Figuur 3: Timer0

```
26 #define TIMERTOP0 100
```

Figuur 4: Timertop

Tijdens iedere cyclus wordt gekeken of de huidige tellerwaarde (timer0) kleiner is dan of gelijk aan de gewenste PWM-instelling (pwm).

- Als $\text{timer0} \leq \text{pwm}$ wordt de bijbehorende uitgangspin hoog gezet, waardoor de ventilator wordt aangestuurd.
- Als $\text{timer0} > \text{pwm}$ wordt de uitgang laag gezet.

```
234     if(timer0 <= pwm)
235     {
236         GPIO_PinWrite(POORT0, PIN_OUT, 1);
237     }
238     else
239     {
240         GPIO_PinWrite(POORT0, PIN_OUT, 0);
241     }
242 }
```

Figuur 5: PWM Aansturing

Hierdoor ontstaat een PWM-signal met een vaste frequentie (bepaald door de timerinstelling) en een variabele duty cycle (bepaald door de waarde van pwm). De duty cycle kan daarbij variëren tussen 0% en 100%, afhankelijk van de gewenste ventilatorsnelheid.

De afstand van de pingpongbal tot de sensor wordt gemeten met een ultrasoensor. Hierbij wordt gebruikgemaakt van de interne hardwaretimer (CTIMER) van de microcontroller als basis voor de tijdsmeting.

De werking is als volgt:

- De ultrasoensor stuurt periodiek automatisch een meetpuls uit.
- De terugkerende echo wordt via een digitale ingang (PIN_IN) van de microcontroller gemeten.
- Op het moment dat het echo-signaal hoog wordt, start een softwarematige telling. Dit gebeurt door tijdens elke timer-interrupt de variabele echo met 1 te verhogen zolang het echo-signaal hoog blijft.

- Zodra het echo-singaal weer laag wordt, stopt de telling. De totale waarde van echo geeft daarmee een maat voor de tijd die het ultrasoonsignaal onderweg is geweest.

```

215         if(GPIO_PinRead(POORT0, PIN_IN))
216     {
217         echo++;
218         startEchoTimer = true;
219     }
220
221     else if(startEchoTimer)
222     {
223         distance = ((echo *10 / 58) - 30) * 0.9;
224         if (distance > 100) {
225             distance = 100;
226         }
227         if (distance < 0) {
228             distance = 0;
229         }
230         echo = 0;
231         startEchoTimer = false;
232     }
233

```

Figuur 6: Ultrasoont aansturing

Omdat de hardwaretimer (CTIMER) op een vaste frequentie draait, levert het aantal getelde echo-stappen een indirecte meting op van de tijdsduur van het ultrasoonsignaal. Deze wordt vervolgens omgerekend naar een afstand in centimeters met behulp van de volgende formule:

$$distance = ((echo *10 / 58) - 30) * 0.9;$$

De microcontroller maakt gebruik van een UART-verbinding om data te versturen en ontvangen via de seriële poort. Dit wordt gebruikt voor zowel het doorgeven van meetdata als voor het instellen van parameters tijdens de uitvoering.

Tijdens de uitvoering wordt meetdata via UART verstuurd naar een externe ontvanger (bijvoorbeeld een pc of een datalogger). Hiervoor wordt in de code gebruik gemaakt van de functie LPUART_WriteBlocking(). Hierbij wordt in de software een tekstregel opgebouwd met de actuele meetwaarden (zoals afstand, PWM-waarde en setpoint), die vervolgens via de UART wordt verzonden.

De functie sendThreeIntsUART() en sendSixIntsUART() zorgen voor het formatteren en verzenden van de meetwaarden:

```

105 void sendThreeIntsUART(int i1, int i2, int i3)
106 {
107 //     char buffer[48];
108 //     int len = sprintf(buffer, "%d,%d,%d\r\n", i1, i2, i3);
109 //     LPUART_WriteBlocking(DEMO_LPUART, (uint8_t *)buffer, len);
110     char buffer[16];
111     int len = sprintf(buffer, "%d\r\n", i1);
112     LPUART_WriteBlocking(DEMO_LPUART, (uint8_t *)buffer, len);
113
114 }
115
116 void sendSixIntsUART(int i1, int i2, int i3, int i4, int i5, int i6)
117 {
118     char buffer[64];
119     int len = sprintf(buffer, "%d,%d,%d,%d,%d,%d\r\n", i1, i2, i3, i4, i5, i6);
120     LPUART_WriteBlocking(DEMO_LPUART, (uint8_t *)buffer, len);
121 }
122

```

Figuur 7: Uart verzending

Hiermee wordt de volledige tekstbuffer in één keer naar de UART-uitgang gestuurd.

Naast het verzenden van meetdata wordt de UART-verbinding ook gebruikt om tijdens het uitvoeren nieuwe setpoints door te sturen naar de microcontroller. Dit gebeurt door het uitlezen van binnenkomende bytes via de functie LPUART_ReadByte() zodra er nieuwe data beschikbaar is:

```
268     // Check of een byte binnen is
269     if (LPUART_GetStatusFlags(DEMO_LPUART) & kLPUART_RxDataRegFullFlag)
270     {
271         uint8_t ch = LPUART_ReadByte(DEMO_LPUART);
272
273         if (ch == '\r' || ch == '\n')
274         {
275             rdbuf[index] = '\0';
276             setpoint = atoi(rdbuf); // omzetten naar int
277             index = 0; // buffer resetten
278         }
279         else if (index < sizeof(rdbuf) - 1)
280         {
281             rdbuf[index++] = ch;
282         }
283     }
284
285
```

Figuur 8: Uart lezing

Ontvangen tekens worden opgeslagen in een tijdelijke buffer (rdbuf). Zodra een complete regel ontvangen is (herkend aan het \r of \n karakter), wordt de tekst omgezet naar een geheel getal (int) met atoi(). Dit nieuwe setpoint wordt vervolgens toegepast in het regelsysteem.

Vervolgens kan binnen de daaropvolgende if-statement de regelcode voor de PID- of AI-aansturing worden uitgevoerd, waarbij de variabelen distance en pwm als invoer en uitvoer dienen.

```
245     if(timer2Expired)
246     {
247         timer2Expired = false;
248         // hoogte = distance / 1000;
249         // setpointfloat = setpoint / 10000;
250         // tout = setpointfloat - hoogte;
251         // fout_integratie += fout * Ts;
252         // fout_afgeleide = (fout - vorige_fout) / Ts;
253         // vorige_fout = fout;
254         //
255         float channels[5] = { setpointfloat, hoogte, fout, fout_integratie, fout_afgeleide };
256         //
257         tss_reg_predict(channels, target);
258         //
259         int AIPWM = (int)target[0]*100;
260         //
261         if (AIPWM < 0) AIPWM = 0;
262         if (AIPWM > 100) AIPWM = 100;
263         //
264         pwm = AIPWM;
265         sendThreeIntsUART(filtered_measurement, (pwm*10), setpoint);
266
267     }
```

Figuur 9: Uitvoerende code

De microcontroller beschikt nu over werkende basiscode, waarin eenvoudig een AI-model geïntegreerd kan worden.

4.2.4: LSTM EN COMPLEX AI

Om te begrijpen hoe AI's werken, is eerst uitgebreid literatuuronderzoek gedaan naar de werking van machine learning en het effect van bijvoorbeeld het aantal neuronen op een model. Op basis hiervan zijn testcodes opgesteld (terug te vinden in de Bitbucket), waarmee is geëxperimenteerd hoe een AI-model opgebouwd kan worden. Het ontwikkelen van een werkende en complexe AI is stapsgewijs verlopen: naarmate de kennis toenam, zijn steeds verdere uitbreidingen doorgevoerd. Deze stappen worden in de onderstaande paragrafen toegelicht.

4.2.4.1: QT CREATOR

Aan het begin van deze sprint is er eerst gekeken naar een bestaande PID-simulatie uit DIS10: de *TempWeerstand*, een temperatuurregelaar die het gedrag van een verwarmingssysteem simuleert. Hieruit is een dataset gegenereerd en opgeslagen als CSV-bestand.

Met behulp van een bestaande trainingscode in Visual Studio Code kon vervolgens een AI-model getraind worden op basis van deze simulatie. Hiervoor is een eenvoudige PyTorch-code gebruikt.

Het doel is om een neuraal netwerk te trainen dat leert hoe een PID-regelaar het verwarmingssysteem aanstuurt. Het AI-model krijgt hierbij als input de actuele toestand van het systeem, en voorspelt de benodigde PWM-waarde (pulsbreedtemodulatie) om de temperatuur richting het gewenste setpoint te sturen.

De dataset die gebruikt wordt bevat waarden zoals:

- de gewenste temperatuur (setpoint),
- de gemeten temperatuur,
- de fout tussen gewenst en gemeten (error),
- de berekende P-, I- en D-term van de PID-regelaar,
- en de gegenereerde PWM op dat moment.

Ons doel is: de AI trainen om dezelfde (of betere) PWM-beslissingen te nemen, zonder dat hij weet *hoe* de PID werkt, maar puur door te leren van de data.

Eerst laden we de dataset (data.csv) en bereiden we de invoer voor. Daarnaast voegen we twee nuttige nieuwe kolommen toe:

1. **Error_prev** – de fout van de vorige stap.
2. **PWM_prev** – de PWM-uitgang van de vorige stap.

Dit doen we omdat een AI, net als een PID-regelaar, beter presteert als hij weet hoe de fout en de actie zich in het verleden ontwikkelden. Zo kan hij tijdsafhankelijke dynamiek leren, zoals overshoot vermijden of stabiel afremmen.

```
import pandas as pd

# Laad de dataset
df = pd.read_csv("data.csv")

# Voeg vorige fout en vorige PWM toe
df["Error_prev"] = df["Error"].shift(1).fillna(0)
df["PWM_prev"] = df["PWM"].shift(1).fillna(0)
```

Figuur 10

Waarom?

- Een gewone PID gebruikt ook voorgaande informatie via de I- en D-term. We geven de AI vergelijkbare context, zodat hij *gedrag* leert in plaats van alleen momentopnames.

Voor goede prestaties moeten de waarden van alle inputs en outputs op een vergelijkbare schaal liggen. We gebruiken standaardisatie (mean = 0, std = 1) via StandardScaler

Zonder normalisatie kunnen grote waarden (zoals PWM = 1000) de AI domineren, waardoor het model moeilijk leert. Door te schalen leert het model gelijkmatig van alle inputdimensies.

de dataset wordt gesplitst in een trainingsset (80%) en een testset (20%). Zo kunnen we het model later testen op ongeziene data.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
```

Figuur 11

PyTorch gebruikt tensor objecten in plaats van standaard arrays. We converteren daarom de NumPy-arrays naar torch.tensor objecten.

```
import torch

X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_test = torch.tensor(y_test, dtype=torch.float32)
```

Figuur 12

Het netwerk heeft 8 ingangen (features), en één uitgang (PWM). We gebruiken een **feedforward neuraal netwerk** met twee verborgen lagen en ReLU-activatiefuncties om niet-lineair gedrag te kunnen leren.

```
import torch.nn as nn

class PID_AI(nn.Module):
    def __init__(self):
        super(PID_AI, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(8, 64), # 8 inputs → 64 neurons
            nn.ReLU(),
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, 1) # Output: PWM
        )

    def forward(self, x):
        return self.model(x)
```

Figuur 13

Waarom deze structuur?

- **64 en 32 neurons** geven het model genoeg capaciteit om complexe relaties te leren.
- **ReLU** voorkomt dat het model "vlakke" zones leert zoals bij sigmoid/tanh.
- **1 output** omdat we slechts één voorspelling willen: de PWM.

We gebruiken Mean Squared Error als verliesfunctie (goed voor regressie) en Adam als optimizer (snelle en stabiele leermethode).

Waarom 500 epochs? Dat is vaak voldoende voor convergentie zonder overfitting. Je kunt dit verhogen als het model nog niet stabiel is.

Stap	Wat gebeurt er?	Waarom is dit belangrijk?
Dataset voorbereiden	Voeg vorige fout en actie toe	Tijdsdynamiek zonder tijd
Normaliseren	Brengt waarden op gelijke schaal	Betere training
Model bouwen	8 input → 64 → 32 → 1 output	Neuraal netwerk voor regressie
Trainen	MSE + Adam optimizer	PWM leren voorspellen
Opslaan	TorchScript .pt	Voor integratie in C++

Tabel 1

Resultaten

Na het trainen van het AI-model werd onderzocht of integratie van de AI mogelijk was binnen Qt Creator. Uit deze poging bleek dat Qt Creator dit niet standaard ondersteunt: voor integratie waren specifieke versies van de compiler en extra build-tools vereist die enkel toegankelijk waren met beheerdersrechten. Hierdoor was het niet mogelijk om de AI direct binnen Qt Creator te gebruiken.

Daarom is besloten om over te stappen naar een eigen simulatie- en validatieomgeving in Python-code. De getrainde AI kon hierdoor niet verder worden ingezet, maar de bestaande trainingscode vormt wel een solide basis voor verdere ontwikkeling. Deze code kan in de toekomst worden uitgebreid en verfijnd, bijvoorbeeld met een betere dataset, aangepaste netwerkarchitectuur of door het model geschikt te maken voor embedded implementatie.

4.2.5: BINNEN VS CODE TESTEN

Om de AI die getraind kan worden doormiddel van de training code te kunnen valideren of testen is er een extra test code die 20% van de dataset pakt en kijkt hoe goed de AI het PWM kan voorspellen.

```

93 # Plot training loss
94 plt.plot(losses)
95 plt.xlabel("Epochs")
96 plt.ylabel("Loss")
97 plt.title("Trainingsverlies")
98 plt.show()
99
100 # Model opslaan
101 torch.save(model.state_dict(), "ai_model.pth")
102 print("Model opgeslagen als ai_model.pth")
103
104 # Evaluatie
105 model.eval()
106 with torch.no_grad():
107     x_test_tensor = torch.FloatTensor(X_test)
108     y_pred = model(x_test_tensor).numpy()
109
110 y_test_original = scaler_y.inverse_transform(y_test.reshape(-1, 1))
111 y_pred_original = scaler_y.inverse_transform(y_pred)
112
113 print(f"Min voorspelde PWM: {y_pred_original.min()}, Max voorspelde PWM: {y_pred_original.max()}")
114 print(f"Min werkelijke PWM: {y_test_original.min()}, Max werkelijke PWM: {y_test_original.max()}")
115

```

Figuur 14

```

116 # Detail plot
117 plt.figure(figsize=(10, 5))
118 plt.plot(y_test_original[500:700], label="Echte PWM", linestyle="dashed", color="red")
119 plt.plot(y_pred_original[500:700], label="Voorspelde PWM", color="blue")
120 plt.xlabel("Sample")
121 plt.ylabel("PWM")
122 plt.legend()
123 plt.title("AI PWM-voorspelling vs. Werkelijke waarden (Detail)")
124 plt.show()
125
126 # Volledige plot
127 plt.figure(figsize=(10, 5))
128 plt.plot(y_test_original, label="Echte PWM", linestyle="dashed", color="red")
129 plt.plot(y_pred_original, label="Voorspelde PWM", color="blue")
130 plt.xlabel("Sample")
131 plt.ylabel("PWM")
132 plt.legend()
133 plt.title("AI PWM-voorspelling vs. Werkelijke waarden")
134 plt.show()

```

Figuur 15

De code maakt twee grafieken:

1. Loss per epoch (trainingsverlies):

Deze grafiek laat zien hoe het verlies (de "loss") van het model verandert tijdens het trainen. Elke epoch is één volledige doorloop van de trainingsdataset. De loss geeft aan hoe goed of slecht het model presteert bij het voorspelen van de juiste waarden tijdens training.

Een dalende curve betekent dat het model steeds beter leert en patronen in de dataset herkent. Hoe sneller de loss daalt, hoe sneller het model de onderliggende verbanden leert.

2. Validatie op 20% van de dataset (testresultaten):

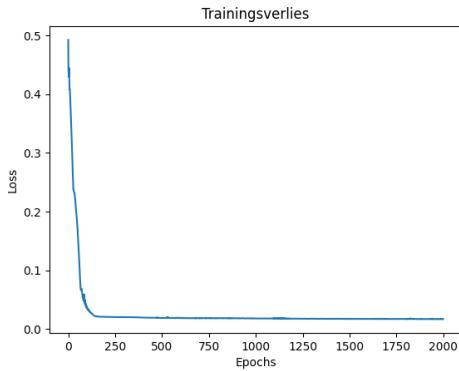
Na het trainen wordt het model getest op een aparte dataset (meestal 20% van de originele data, die niet is gebruikt bij training). Met een Torch-functie (zoals `model.eval()` gevolgd door `with torch.no_grad()`) wordt de AI geactiveerd om voorspellingen te doen op deze testdata.

Voor elke invoer (bijv. een bepaalde sensorwaarde of afstand) voorspelt het model een PWM-output. Deze voorspellingen worden vergeleken met de echte PWM-waarden. De resultaten worden gevisualiseerd in een grafiek, zodat je kunt zien hoe nauwkeurig het model op ongeziene data presteert.

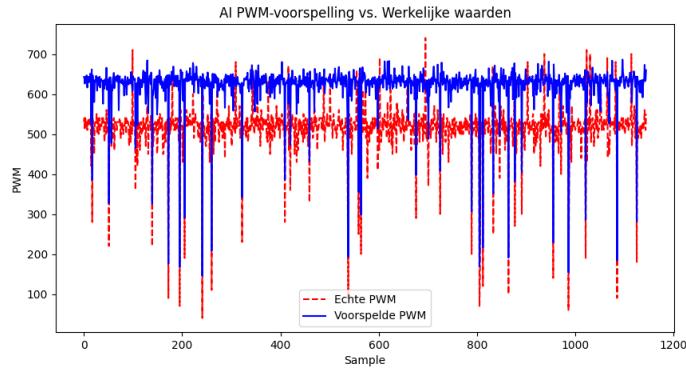
4.2.5.1: EERSTE RESULTATEN

Alle resultaten van de tussen testen van verschillende soorten modellen, neuronen en lagen zijn te vinden in de bijlage.

Dit waren de resultaten van de AI die eerst was gebruikt voor QT creator maar getest binnen VScode.



Figuur 16: Trainingsverlies



Figuur 17: voorspelling vs werkelijk

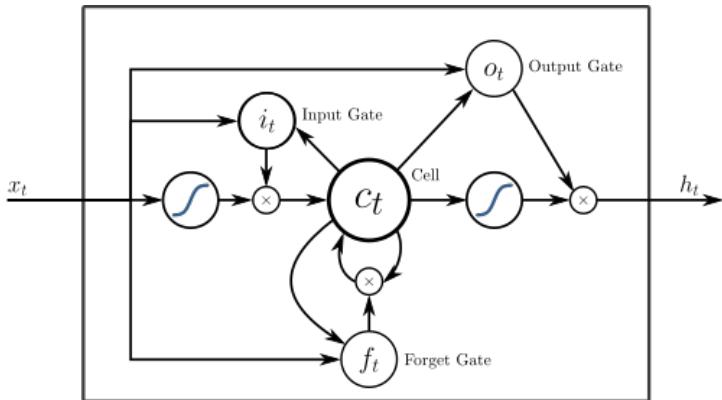
Je ziet dat het model vrij soepel door de dataset heen loopt, met een snel dalende loss tijdens de epochs. Dit geeft aan dat het model snel leert patronen te herkennen in de trainingsdata. Echter, uit de validatie bleek dat de voorspelde waarden aanzienlijk afwijken van de werkelijke PWM-waarden. Met andere woorden: hoewel het model goed lijkt te trainen, generaliseert het slecht naar nieuwe, ongeziene data.

Op basis van deze bevindingen zijn er twee verschillende benaderingen gekozen om verder te gaan. Er is geëxperimenteerd met verschillende datasets, modelstructuren en variaties in de opbouw van de AI — waaronder het aantal lagen en neuronen per laag.

De exacte instellingen en bijbehorende resultaten van deze experimenten zijn te vinden in de bijlage.

4.2.5.2: LSTM

Na het proberen van verschillende mogelijkheden, en een kort literatuuronderzoek, werd gekozen voor een LSTM-model. LSTM staat voor Long Short-Term Memory en is een speciaal type Recurrent Neural Network (RNN) dat ontwikkeld is om sequentiële data te verwerken, oftewel gegevens waarbij de volgorde en het tijdsverloop belangrijk zijn.



Figuur 18

Wat is een LSTM?

Een traditioneel neuraal netwerk (zoals een feedforward netwerk) behandelt elke input onafhankelijk. Dat werkt goed voor statische data, maar minder goed voor situaties waarbij het huidige gedrag afhankelijk is van eerdere waarden — bijvoorbeeld bij tijdreeksen, bewegingen of signalen die over tijd veranderen.

LSTM-netwerken zijn hier juist wél goed in. Ze bevatten interne geheugencellen die in staat zijn om belangrijke informatie over langere tijd vast te houden (long-term memory), terwijl irrelevante informatie wordt weggefilterd. Dit gebeurt via zogeheten gates:

- **Forget gate:** bepaalt welke informatie uit het geheugen verwijderd moet worden.
- **Input gate:** beslist welke nieuwe informatie wordt toegevoegd aan het geheugen.
- **Output gate:** bepaalt welke informatie uit het geheugen als output wordt gebruikt.

Deze structuur voorkomt dat het model last krijgt van problemen zoals het vergeten van oude informatie (wat bij gewone RNN's vaak gebeurt, het zogenaamde "vanishing gradient"-probleem).

Waarom LSTM goed past bij deze toepassing

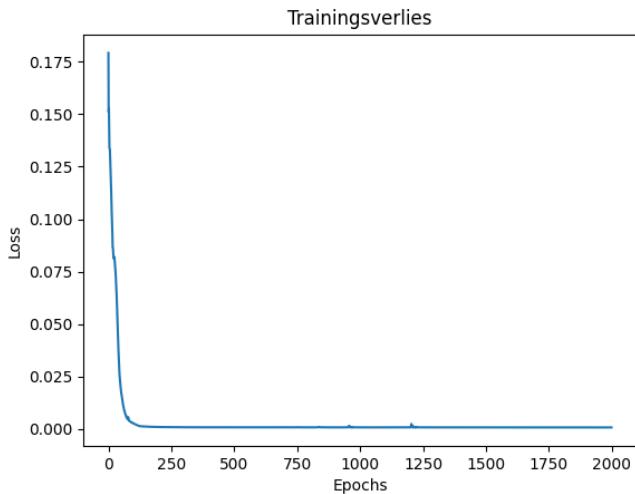
In dit project wordt een AI-model getraind om op basis van een reeks meetwaarden (bijvoorbeeld afstandsmetingen) de juiste PWM-waarde te voorspellen. Omdat deze waarden niet op zichzelf staan, maar een tijdsafhankelijk verloop hebben, is het belangrijk dat het model begrijpt hoe eerdere inputs samenhangen met de huidige situatie.

LSTM's zijn bij uitstek geschikt voor dit soort toepassingen. Ze kunnen de relatie over tijd herkennen — bijvoorbeeld hoe een verandering in afstand in opeenvolgende metingen leidt tot een andere aansturing — en die informatie meenemen in de voorspelling. Daardoor zijn ze beter in staat om stabiele en nauwkeurige output te leveren in situaties met dynamische of fluctuerende invoer.

In de praktijk bleek het LSTM-model veelbelovend, omdat het in staat was om zowel tijds patronen als opeenvolgende afhankelijkheden in de dataset te herkennen, iets waar eenvoudige feedforward netwerken vaak moeite mee hebben.

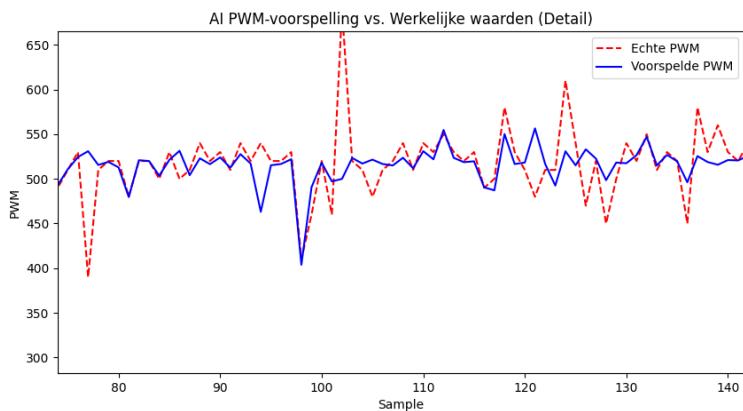
De instellingen van het LSTM-model, zoals het aantal geheugencellen, lagen, activatiefuncties en trainingsparameters, zijn opgenomen in de bijlage, samen met de testresultaten en vergelijkingen met eerdere modellen.

4.2.5.3: RESULTATEN VAN LSTM



Figuur 19

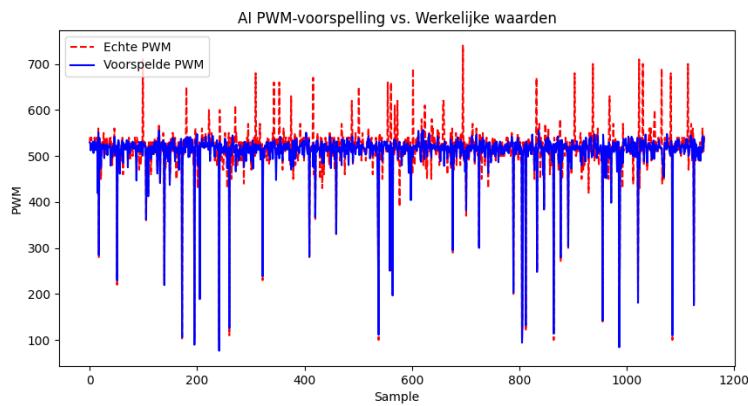
Deze figuur toont het verloop van het trainingsverlies (loss) tijdens het trainen van het model. Al binnen de eerste 200 epochs daalt het verlies aanzienlijk en nadert het vrijwel nul. Dit betekent dat het model zeer snel leert en de trainingsdata nauwkeurig weet te benaderen. Na verloop van tijd stabiliseert de loss-waarde, wat aangeeft dat het model convergeert.



Figuur 20

Deze figuur toont een kleinere tijdsrange van het signaal, van sample 75 tot 140, waardoor de details beter zichtbaar zijn. Hieruit blijkt dat de voorspelde waarden van het model sterk overeenkomen met de echte PWM. De fluctuaties worden niet uitgevlakt, maar juist goed gevuld — zelfs wanneer de PWM tijdelijk sterk daalt of stijgt.

De voorspelde lijn (blauw) volgt de vorm van de werkelijke lijn (rood) met relatief kleine afwijkingen, en dit geldt ook voor de scherpe pieken naar beneden die aanwezig zijn in beide signalen.



Figuur 21

In deze figuur zijn de voorspelde PWM-waarden (blauw) uitgezet tegenover de werkelijke PWM-waarden (rood, gestippeld) over de gehele dataset. Wat opvalt is dat het model de algemene vorm en de meeste fluctuaties goed volgt. De voorspelling blijft niet constant, maar beweegt actief mee met de werkelijke PWM, inclusief scherpe pieken en dalen.

Hoewel er op enkele punten een kleine afwijking zichtbaar is, weet het model zelfs de plotselinge dalingen en stijgingen in PWM redelijk accuraat te volgen, wat aantoont dat het model ook gevoelige dynamiek in de data heeft opgepikt.

CONCLUSIE

Het LSTM-model toont goede prestaties in zowel trendherkenning als detailnauwkeurigheid. De lage loss-waarde duidt op een effectieve training, en de vergelijkingen tussen voorspellingen en echte PWM-waarden laten zien dat het model in staat is om zowel langzame trends als plotselinge fluctuaties betrouwbaar te voorspellen.

Sterke punten:

- Herkent zowel trends als pieken/dalen in het signaal.
- Reageert actief op veranderingen in de invoer.
- Lage training loss, snelle convergentie.

Verbeterpunten voor toekomstig werk:

- Nauwkeurigheid op enkele extreme punten kan nog worden geoptimaliseerd.
- Validatie op nieuwe data (bijv. unseen testsets) is nodig om generalisatie te bevestigen.

Dit was ook het einde van de sprint en werd de verdere ontwikkeling overgedragen.

4.3: SPRINT 3 – AI-MODELEN EN SIMULATIE

In Sprint 3 zijn we gestart met het bouwen van de eerste AI-modellen. Het team is hierbij opgesplitst:

- Piet en Yered gingen aan de slag met de NXP eIQ Time Series Studio, om te onderzoeken of deze toolkit een geschikte route bood voor embedded AI.
- Mitchel en Thomas kozen ervoor een eigen AI-model te ontwikkelen,
- Mitchel is aan de slag gegaan met het moduleren van de ping pong opstelling en heeft hierbij een simulatie gemaakt van een PID-regelaar.

In deze sprint zijn daarnaast twee onderzoeken gestart. De uitwerking hiervan is terug te vinden in het hoofdstuk '[Onderzoek](#)'.

4.3.1 SIMULATIE

Tijdens deze sprint is in Python een simulatie ontwikkeld van de pingpongopstelling, met als doel het gedrag van de pingpongbal te modelleren en de aansturing te optimaliseren. De simulatie is gebaseerd op de fysische eigenschappen van de bal en de interactie met de omgeving, zoals luchtweerstand en zwaartekracht. Door verschillende scenario's te simuleren, konden we een representatieve dataset genereren voor het trainen van het neurale netwerk.

Een belangrijke uitdaging was het realiseren van een realistische simulatie die de werkelijke opstelling zo nauwkeurig mogelijk benadert, inclusief de integratie van een PID-regelaar. Parameters als de massa van de bal, zwaartekracht en PID-instellingen zijn zorgvuldig afgestemd om het systeemgedrag correct te modelleren. Hierdoor ontstond een dataset die geschikt is voor het trainen van het neurale netwerk om het gedrag van de PID-regelaar na te bootsen.

De simulatie is geprogrammeerd in Python, waarbij gebruik is gemaakt van NumPy en Matplotlib voor wiskundige berekeningen en visualisaties. Dit maakte het mogelijk om snel te experimenteren en de simulatie aan te passen op basis van de resultaten. Centraal in de simulatie staat een Python-implementatie van een PID-regelaar, die de hoogte van de bal regelt door de ventilator aan te sturen op basis van het verschil tussen de gewenste en gemeten hoogte.

Deze aanpak bood twee belangrijke voordelen: enerzijds konden we het systeemgedrag analyseren en optimaliseren zonder direct op de hardware te werken, anderzijds leverde het een dataset op waarmee het neurale netwerk getraind kon worden. Hierdoor is het netwerk voorbereid op uiteenlopende situaties en kan het zorgen voor een betrouwbare aansturing van de pingpongbal.

Om het gedrag van de pingpongbal goed te simuleren, zijn de krachten die op de bal werken gemodelleerd. Dit zijn voornamelijk de zwaartekracht en de luchtkracht (lift) die door de ventilator wordt opgewekt. De verticale krachten op de bal worden als volgt beschreven:

- Luchtkracht(lift):

$$F_{lucht} = C \times u^2$$

- Zwaartekracht:

$$F_{zwaartekracht} = m \times g$$

- Resulterende kracht en versnelling:

De totale kracht is het verschil tussen de opwaartse en neerwaartse kracht. De resulterende versnelling van de bal wordt dan:

$$F_{netto} = C \times u^2 - m \times g$$

In de simulatiecode wordt deze laatste formule direct toegepast, waarbij de constante (C) zodanig gekozen is dat deze al gedeeld is door de massa (m), zodat de versnelling eenvoudig als volgt berekend kan worden:

```
a = C * u**2 - g
```

Deze krachten vormen de basis voor het simulatiemodel van de beweging van de pingpongbal. Daarnaast om de hoogte van de pingpongbal te regelen, is gebruik gemaakt van een PID-regelaar (Proportioneel–Integrerend–Differentiërend). De PID-

regelaar stuurt de ventilator aan op basis van het verschil tussen de gewenste hoogte (setpoint) en de werkelijke hoogte van de bal. De algemene vorm van de PID-regeling is:

$$u(t) = K_p \cdot e(t) + K_i \cdot \int e(t) dt + K_d \cdot \frac{de(t)}{dt}$$

Waarbij:

- $u(t)$: de uitgestuurde waarde naar de ventilator (PWM-singaal)
- $e(t)$: de fout = gewenste hoogte - gemeten hoogte
- K_p : proportionele versterking
- K_i : integrale versterking
- K_d : differentiële versterking

In de simulatie wordt dit als volgt geïmplementeerd in code:

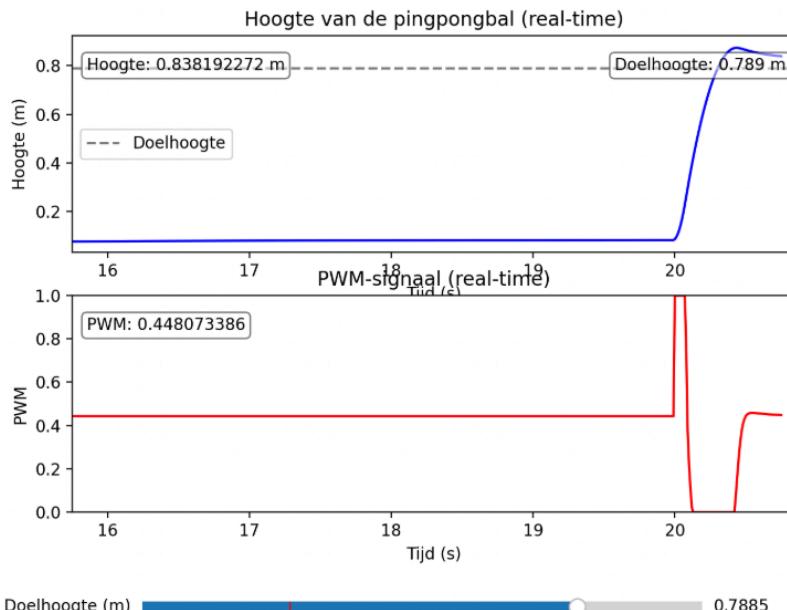
```
e = h_target - h           # fout
e_sum += e * dt            # integratie van de fout
de = (e - e_prev) / dt    # afgeleide van de fout
u = Kp * e + Ki * e_sum + Kd * de
```

Vervolgens wordt de output begrensd tussen 0 en 1:

```
u = max(0.0, min(1.0, u))
```

Deze berekende waarde van u wordt gebruikt om de kracht van de ventilator te bepalen en zo de bal op de gewenste hoogte te houden. De PID-regelaar is essentieel om snelle, stabiele en nauwkeurige hoogtecorrecties mogelijk te maken.

Daarnaast voor het simuleren van de beweging van de pingpongbal is een tijdstap van 0.01 seconden gebruikt. Dit zorgt voor een voldoende nauwkeurige simulatie van de beweging, waarbij de positie en snelheid van de bal elke 0.01 seconden worden bijgewerkt op basis van de krachten die op de bal werken. Hieronder een voorbeeld van de simulatie.



Na dat dit gegenereerd wordt kan de data geëxporteerd worden naar een CSV bestand zie afbeelding hieronder.

Tijd (s)	Setpoint (m)	Hoogte (m)	Fout	Fout_Integratie	Fout_Afgeleide	PWM
0.000000	0.700171	0.004019	0.700171	0.007002	70.017086	1.000000
0.010000	0.700171	0.012957	0.696152	0.013963	-0.401900	1.000000
0.020000	0.700171	0.024114	0.688114	0.020844	-0.803800	1.000000
0.030000	0.700171	0.040190	0.676057	0.027605	-1.205700	1.000000
0.040000	0.700171	0.060285	0.659981	0.034205	-1.607600	1.000000
0.050000	0.700171	0.084399	0.639886	0.040604	-2.009500	1.000000
0.060000	0.700171	0.112532	0.615772	0.046761	-2.411400	1.000000
0.070000	0.700171	0.143472	0.587639	0.052638	-2.813300	0.870446
0.080000	0.700171	0.174317	0.556698	0.058205	-3.094038	0.420767

4.4: SPRINT 4 – DATASET UITBREIDEN, VERFIJNEN EN IMPORTEREN MODEL OP DE MICROCONTROLLER

In Sprint 4 werd de dataset verder uitgebreid. Naast de meetdata uit de pingpongopstelling en de simulatie werden extra datasets verzameld onder verschillende condities. Hierdoor kon het trainingsmateriaal worden verbeterd en kon het model beter generaliseren. Ook is in deze fase geëxperimenteerd met datavoorbewerking, zoals filtering en normalisatie, om de inputdata geschikt te maken voor effectieve training.

- Voor de implementatie is Mitchel bezig geweest met het importeren van een model in de microcontroller, Hierbij is een model gemaakt die een waarde van de sinus voorspelt.

Daarnaast is in deze sprint verder gewerkt aan de ontwikkeling van beide modellen.

4.4.1 IMPORTEREN MODEL OP DE MICROCONTROLLER

In deze sprint hebben we een simpel neuraal netwerk geïmplementeerd op de microcontroller en getest of het netwerk correct kon worden geladen en uitgevoerd. Hiervoor hebben we gebruik gemaakt van TensorFlow Lite, een versie van TensorFlow die speciaal is ontworpen voor embedded systemen. Het doel was om het getrainde netwerk te importeren in de microcontroller en te testen of het netwerk correct kon functioneren. Om te bevestigen dat het importeren van een neuraal netwerk op de microcontroller goed ging, hebben we een eenvoudig neuraal netwerk geïmplementeerd dat een sinusgolf kan voorspellen. Dit netwerk was veel eenvoudiger dan het uiteindelijke netwerk dat het gedrag van de PID-regelaar moest nabootsen, maar het diende als een proof of concept. We hebben de stappen doorlopen om het netwerk te converteren naar TensorFlow Lite-formaat en dit op de microcontroller te laden. Voor het doorlopen van deze stappen hebben we de officiële TensorFlow Lite-documentatie geraadpleegd, die gedetailleerde instructies biedt voor het converteren van een TensorFlow-model naar TensorFlow Lite en het implementeren op een microcontroller. Hieronder staan de stappen die we hebben doorlopen:

1. **Model Training:** We hebben een eenvoudig neuraal netwerk getraind met tensorflow in python, dat een sinusgolf voorspelt.
2. **Model Export:** Het getraind model is geëxporteerd naar het Tensorflow Lite formaat met behulp van de `tf.lite.TFLiteConverter`.
3. **Implementatie op Microcontroller:** Het geoptimaliseerde TensorFlow Lite-model is geïmporteerd in de microcontrolleromgeving, waarbij we gebruik hebben gemaakt van de TensorFlow Lite Micro-bibliotheek. Deze fase was cruciaal om te begrijpen hoe we de complexiteit van het uiteindelijke netwerk konden reduceren zonder in te boeten op de prestaties. Door deze proof of concept hebben we waardevolle ervaring opgedaan met het werken met TensorFlow Lite en de beperkingen van de microcontroller, wat ons voorbereidde op de volgende stappen in het project.

4.4.1.1 MODEL TRAINING

Voor het trainen van het neurale netwerk hebben we een eenvoudig model gebruikt dat een sinusgolf kan voorspellen. Dit model is getraind met behulp van TensorFlow in Python. De stappen voor het trainen van het model zijn als volgt:

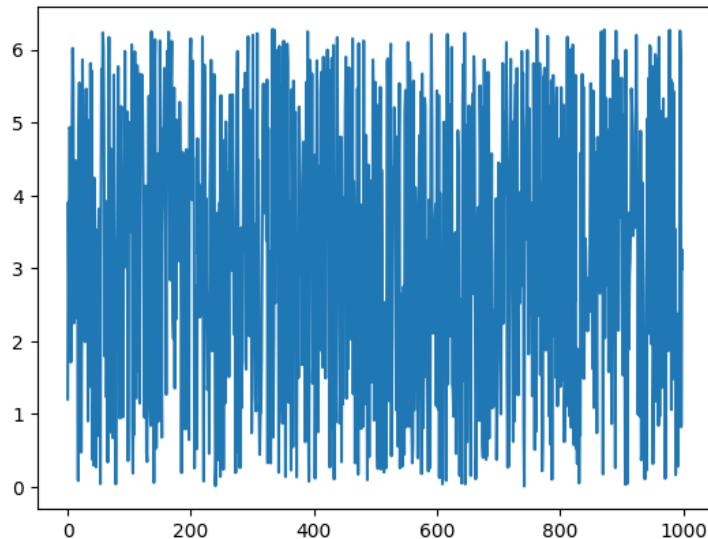
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import math
import keras
from keras import layers

#Numpy 2.1.3
#TensorFlow 2.19.0
#Keras 3.9.2

# Settings
nsamples = 1000      # Number of samples to use as a dataset
val_ratio = 0.2        # Percentage of samples that should be held for validation set
test_ratio = 0.2        # Percentage of samples that should be held for test set
tflite_model_name = 'sine_model' # Will be given .tflite suffix
c_model_name = 'sine_model'      # Will be given .h suffix

np.random.seed(1234)
x_values = np.random.uniform(low=0, high=(2 * math.pi), size=nsamples)
plt.plot(x_values)
```

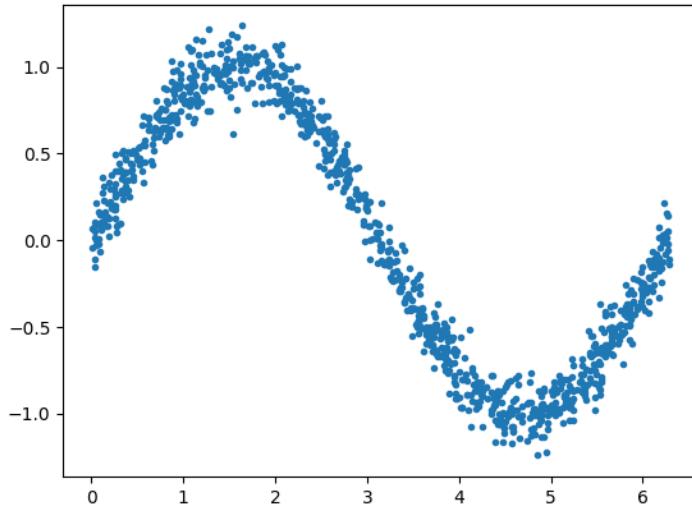
Hier wordt een dataset gegenereerd met willekeurige waarden. Deze waarden worden gebruikt als invoer voor het neurale netwerk. zie afbeelding hieronder voor een voorbeeld van de willekeurige waarden.



Daarna wordt de dataset gegenereerd door de sinusfunctie toe te passen op de willekeurige waarden. Dit wordt gedaan met de volgende code:

```
y_values = np.sin(x_values) + (0.1 * np.random.randn(x_values.shape[0]))
plt.plot(x_values, y_values, '.')
```

Hier wordt de sinusfunctie toegepast op de willekeurige waarden en wordt er wat ruis toegevoegd om de dataset realistischer te maken. Dit is belangrijk omdat het neurale netwerk moet leren om de sinusgolf te voorspellen, zelfs als er wat ruis in de data zit. Zie afbeelding hieronder voor een voorbeeld van de dataset.



Daarna wordt de dataset opgesplitst in een trainingstest, een validatie set en een testset. Dit is belangrijk om ervoor te zorgen dat het neuraal netwerk goed generaliseert en niet alleen de trainingsdata leert. De splitsing wordt gedaan met de volgende code:

```

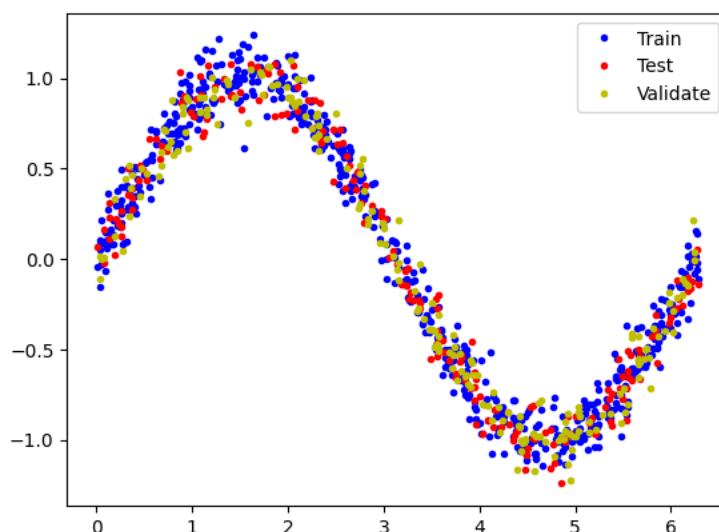
val_split = int(val_ratio * nsamples)
test_split = int(val_split + (test_ratio * nsamples))
x_val, x_test, x_train = np.split(x_values, [val_split, test_split])
y_val, y_test, y_train = np.split(y_values, [val_split, test_split])

# Check that our splits add up correctly
assert(x_train.size + x_val.size + x_test.size) == nsamples

# Plot the data in each partition in different colors:
plt.plot(x_train, y_train, 'b.', label="Train")
plt.plot(x_test, y_test, 'r.', label="Test")
plt.plot(x_val, y_val, 'y.', label="Validate")
plt.legend()
plt.show()

```

Hier worden de willekeurige waarden en de bijbehorende sinuswaarden opgesplitst in drie sets: een trainingstest, een validatie set en een testset. De validatie set wordt gebruikt om het model te optimaliseren tijdens het trainen, terwijl de testset wordt gebruikt om de uiteindelijke prestaties van het model te evalueren. zie afbeelding hieronder van de dataset na het splitsen in de verschillende sets.



Daarna wordt het neuraal netwerk gedefinieerd. Dit wordt gedaan met de volgende code:

```
model = tf.keras.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(1,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1))

model.summary()
```

Hier wordt een neuraal netwerk gedefinieerd met drie lagen. Je hebt eigenlijk vier lagen alleen die verbinden we met de eerste laag van de hidden layer. De eerste twee lagen zijn volledig verbonden lagen met 16 neuronen en een ReLU-activatiefunctie. De laatste laag is een volledig verbonden laag met één neuron, die de voorspelling van de sinusgolf zal doen. Dit model is eenvoudig genoeg om op de microcontroller te draaien, maar krachtig genoeg om de sinusgolf te voorspellen. Daarnaast kan je met de `model.summary()` methode een samenvatting van het model bekijken, inclusief het aantal parameters en de structuur van het netwerk. Dit helpt om te begrijpen hoe het model is opgebouwd en hoeveel rekenkracht er nodig is om het uit te voeren. Zie afbeelding hieronder voor een voorbeeld van de samenvatting van het model.

```
Model: "sequential"



| Layer (type)    | Output Shape | Param # |
|-----------------|--------------|---------|
| dense (Dense)   | (None, 16)   | 32      |
| dense_1 (Dense) | (None, 16)   | 272     |
| dense_2 (Dense) | (None, 1)    | 17      |



Total params: 321 (1.25 KB)

Trainable params: 321 (1.25 KB)

Non-trainable params: 0 (0.00 B)
```

Daarna wordt het model gecompileerd. Dit wordt gedaan met de volgende code:

```
model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
```

Hier wordt het model gecompileerd met de RMSprop-optimizer en de mean squared error (MSE) als verliesfunctie. De MSE is een veelgebruikte verliesfunctie voor regressieproblemen, zoals het voorspellen van de sinusgolf. De mean absolute error (MAE) wordt ook gebruikt als metriek om de prestaties van het model te evalueren. Dit geeft een idee van hoe goed het model presteert op de testset. Daarna wordt het model getraind. Dit wordt gedaan met de volgende code:

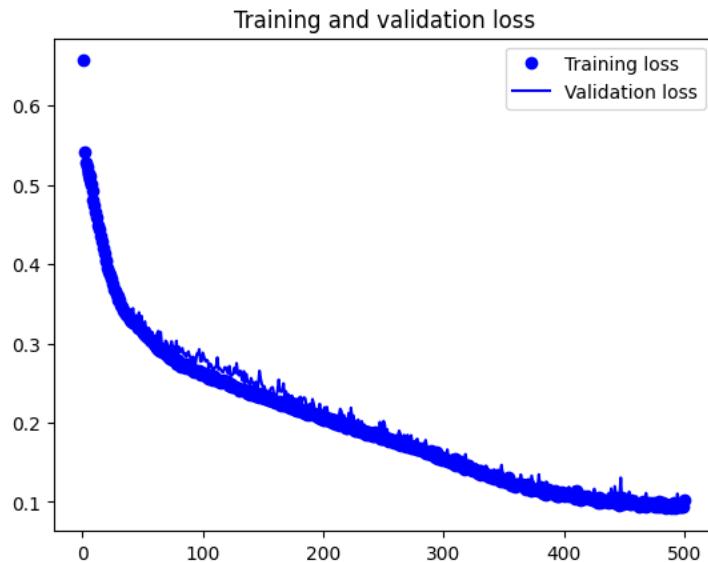
```
history = model.fit(x_train,
                     y_train,
                     epochs=500,
                     batch_size=100,
                     validation_data=(x_val, y_val))

# Plot the training history
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```

Hier wordt het model getraind op de trainingsset met 500 epochs en een batchgrootte van 100. De validatieset wordt gebruikt om het model te optimaliseren tijdens het trainen. De training zal de gewichten van het model aanpassen om de voorspellingen zo nauwkeurig mogelijk te maken. De training kan enige tijd duren, afhankelijk van de grootte van de dataset en de complexiteit van het model. Gedurende de training worden de verlies- en nauwkeurigheidswaarden voor zowel de trainings- als de validatieset bijgehouden. Dit helpt om te controleren of het model niet overfit op de trainingsdata en of het goed generaliseert naar nieuwe data. zie afbeelding hieronder voor een voorbeeld van de training en validatie verlies.

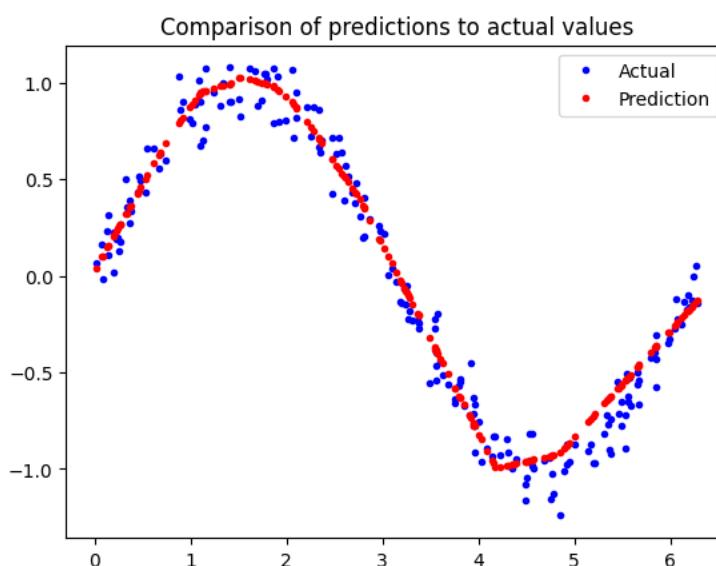


om vervolgens de prestaties van het model te evalueren op de testset, kunnen we de volgende code gebruiken:

```
# Plot predictions against actual values
predictions = model.predict(x_test)

plt.clf()
plt.title("Comparison of predictions to actual values")
plt.plot(x_test, y_test, 'b.', label='Actual')
plt.plot(x_test, predictions, 'r.', label='Prediction')
plt.legend()
plt.show()
```

Hiermee worden de voorspellingen van het model vergeleken met de werkelijke waarden in de testset. Dit geeft een visuele representatie van hoe goed het model presteert en of het in staat is om de sinusgolf nauwkeurig te voorspellen. Zie afbeelding hieronder voor een voorbeeld van de voorspellingen van het model.



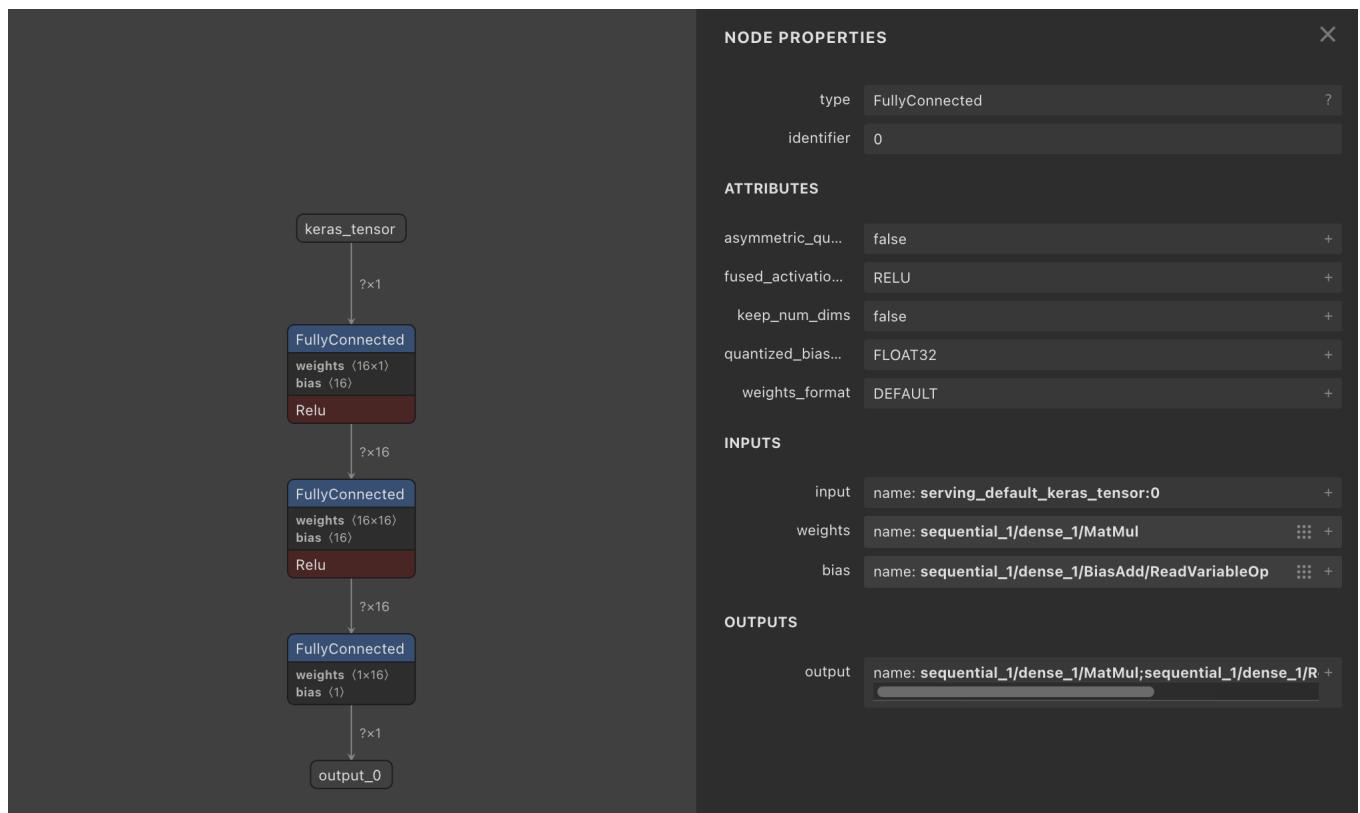
4.4.1.2 MODEL EXPORTEREN

Nu het model is getraind, kunnen we het exporteren naar het TensorFlow Lite-formaat. Dit is een geoptimaliseerde versie van het model die speciaal is ontworpen voor embedded systemen, zoals de FRDM-MCXN947 microcontroller. We gebruiken de volgende code om het model te converteren naar TensorFlow Lite:

```
# Convert Keras model to a tflite model
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]
tflite_model = converter.convert()

open(tflite_model_name + '.tflite', 'wb').write(tflite_model)
```

Hier wordt de TFLiteConverter gebruikt om het Keras-model te converteren naar het TensorFlow Lite-formaat. We hebben ook de optimalisatie voor grootte ingeschakeld, zodat het model zo klein mogelijk wordt gemaakt voor gebruik op de microcontroller. Het geconverteerde model wordt opgeslagen in een bestand met de naam sine_model.tflite. Dit bestand kan vervolgens worden gebruikt op de microcontroller om het neurale netwerk uit te voeren. Je kan vervolgens via een applicatie als "Netron" het model bekijken en controleren of het correct is geconverteerd. Dit is handig om te zien of het model de juiste structuur heeft en of de gewichten correct zijn ingesteld. zie afbeelding hieronder van het model in Netron.



daarna moeten we het model exporteren naar C-code, zodat het op de microcontroller kan worden uitgevoerd. Dit wordt gedaan met de volgende code:

```
xxd -i sine_model.tflite > sine_model.h
```

Let op deze code hierboven moet in de terminal ingevoerd worden in de directory waar de "sine_model.tflite" zich bevindt. Om het TensorFlow Lite-model bruikbaar te maken op de microcontroller, wordt gebruikgemaakt van xxd, een tool die binaire bestanden omzet naar een C-headerbestand. Hiermee wordt het modelbestand (sine_model.tflite) geconverteerd naar een C-bestand (sine_model.h) dat de binaire data van het model bevat als een array. Dit bestand kan direct worden opgenomen in de microcontrollercode, zodat het model eenvoudig geladen en uitgevoerd kan worden op de FRDM-MCXN947. Hieronder zie je een voorbeeld van de inhoud van zo'n C-headerbestand.

```
unsigned char sine_model_tflite[] = {
    0x1c, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x14, 0x00, 0x20, 0x00,
    0x1c, 0x00, 0x18, 0x00, 0x14, 0x00, 0x10, 0x00, 0x0c, 0x00, 0x00, 0x00,
    0x08, 0x00, 0x04, 0x00, 0x14, 0x00, 0x00, 0x00, 0x1c, 0x00, 0x00, 0x00,
    0x98, 0x00, 0x00, 0x00, 0xf0, 0x00, 0x00, 0x00, 0x14, 0x07, 0x00, 0x00,
    0x24, 0x07, 0x00, 0x00, 0x0c, 0x0c, 0x00, 0x00, 0x03, 0x00, 0x00, 0x00,
    0x01, 0x00, 0x00, 0x00, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0xa, 0x00,
    0x10, 0x00, 0x0c, 0x00, 0x08, 0x00, 0x04, 0x00, 0xa, 0x00, 0x00, 0x00,
    0x0c, 0x00, 0x00, 0x00, 0x1c, 0x00, 0x00, 0x00, 0x3c, 0x00, 0x00, 0x00,
    0x0f, 0x00, 0x00, 0x00, 0x73, 0x65, 0x72, 0x76, 0x69, 0x6e, 0x67, 0x5f,
    0x64, 0x65, 0x66, 0x61, 0x75, 0x6c, 0x74, 0x00, 0x01, 0x00, 0x00, 0x00,
    0x04, 0x00, 0x00, 0x00, 0x90, 0xff, 0xff, 0x09, 0x09, 0x00, 0x00, 0x00,
    0x04, 0x00, 0x00, 0x00, 0x08, 0x00, 0x00, 0x00, 0x6f, 0x75, 0x74, 0x70,
    0x75, 0x74, 0x5f, 0x30, 0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00,
    0x04, 0x00, 0x00, 0x00, 0xba, 0xf9, 0xff, 0xff, 0x04, 0x00, 0x00, 0x00,
};

unsigned int sine_model_tflite_len = 3168;
```

4.4.1.3 MODEL IMPLEMENTEREN OP DE MICROCONTROLLER

4.5: SPRINT 5 –ZELF ONTWIKKELD MODEL EN INTEGRATIE

Tijdens Sprint 5 zijn beide AI-modellen daadwerkelijk geïmplementeerd op de hardware. Hierbij liepen we tegen verschillende problemen aan:

- Het zelfontwikkelde model van Mitchel en Thomas gaf een goed resultaat alleen de integratie in de microcontroller is nog niet werkend gekregen.
- Het elQ-gebaseerde model van Piet en Yered gaf problemen met de compatibiliteit van de gegenereerde libraries binnen de embedded omgeving.

Deze implementatiefase bracht belangrijke leerervaringen met zich mee over de praktische integratie van AI-modellen op embedded hardware.

4.5.1: ZELFONTWIKKELDE MODEL

In de laatste sprint hebben we ons gericht op het trainen van een neuraal netwerkmodel en de integratie ervan in de microcontroller. We hebben een neuraal netwerk getraind met behulp van TensorFlow en het model geconverteerd naar een formaat dat geschikt is voor gebruik op een microcontroller. Vervolgens hebben we het model geïntegreerd in onze microcontroller-toepassing, zodat we het kunnen gebruiken voor real-time inferentie. Voor de integratie hebben we het nog niet werkend kunnen krijgen, maar we hebben wel de basis gelegd voor de integratie van het model in de microcontroller. Voor verdere uitleg van het model is er een handleiding geschreven die de stappen beschrijft van het trainen van het model. Zie hieronder voor de stappen:

Benodigdheden

Voordat we aan de slag gaan met het neuraal netwerk, zijn er een aantal benodigdheden die geïnstalleerd moeten worden. Zorg ervoor dat de juiste versie van Python en de benodigde bibliotheken hebt. De volgende onderdelen zijn essentieel voor het ontwikkelen, trainen en evalueren van het neuraal netwerk:

- **Python 3.1 of hoger:** De programmeertaal waarin het neuraal netwerk wordt ontwikkeld. Zorg dat je de nieuwste versie hebt geïnstalleerd.
- **Tensorflow 2.11 of hoger:** Een populaire open-source bibliotheek voor machine learning en deep learning, gebruikt voor het bouwen en trainen van neurale netwerken.
- **Numpy:** Een fundamentele bibliotheek voor wetenschappelijk rekenen in Python, handig voor het werken met arrays en wiskundige functies.
- **Matplotlib:** Een bibliotheek voor het maken van visualisaties in Python, nuttig voor het plotten van resultaten en het visualiseren van data.
- **Pandas:** Hiermee kan je gemakkelijk CSV-data importeren in de python code.
- **Jupyter Notebook:** Een interactieve omgeving voor het schrijven en uitvoeren van Python-code, handig voor het ontwikkelen en testen van het neuraal netwerk.
- **Een teksteditor of IDE:** Een omgeving waarin je Python-code kunt schrijven en uitvoeren, zoals Visual Studio Code, PyCharm of Jupyter Notebook.

Versies van gebruikte pakketten

Voor dit project zijn de volgende pakketversies gebruikt om compatibiliteit en reproduceerbaarheid te waarborgen:

- **Pandas:** 2.2.3 (data-analyse en visualisatie)
- **NumPy:** 1.26.4 (data-analyse en visualisatie)
- **Matplotlib:** 3.10.1 (data-analyse en visualisatie)
- **Tensorflow:** 2.15.0 (voor het bouwen, trainen en evalueren van het neuraal netwerk)
- **Keras:** 2.15.0 (voor het bouwen, trainen en evalueren van het neuraal netwerk)
- **Scikit-learn:** 2.00.0 (dataset splitsing en normalisatie)

Hieronder staat een voorbeeld van de benodigde import in Python. Voeg deze code toe aan het begin van je Python-script of Jupyter Notebook:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from keras import layers, models, callbacks, preprocessing
from sklearn.model_selection import train_test_split
from keras.layers import Input, Dense, Conv1D, Flatten
from keras.models import Model, Sequential
from sklearn.preprocessing import MinMaxScaler
import keras
```

Data voorbereiden:

Voordat het neurale netwerk getraind kan worden moeten we er eerst voor zorgen dat we data hebben om mee te trainen. Voor het aanroepen van die data kan je code van hieronder gebruiken. Let op! Data moet een CSV-bestand zijn en om de directory te krijgen van de CSV klik op "Copy path".

```
# Voorbeeld van een bestandslocatie voor je CSV-data:
CSV_PATH = './data/simulatie_resultaten.csv'
CSV_DATA = pd.read_csv(CSV_PATH)
```

Om als voorbeeld maken we gebruik van de CSV die tijdens dit project is gebruikt om het neurale netwerk te trainen.

Tijd (s)	Setpoint (m)	Hoogte (m)	Fout	Fout_Integratie	Fout_Afgeleide	PWM
0.000000	0.700171	0.004019	0.700171	0.007002	70.017086	1.000000
0.010000	0.700171	0.012057	0.696152	0.013963	-0.401900	1.000000
0.020000	0.700171	0.024114	0.688114	0.020844	-0.803800	1.000000
0.030000	0.700171	0.040190	0.676057	0.027605	-1.205700	1.000000
0.040000	0.700171	0.060285	0.659981	0.034205	-1.607600	1.000000
0.050000	0.700171	0.084399	0.639886	0.040604	-2.009500	1.000000

Om vervolgens de inputs en outputs aan elkaar te koppelen zodat het neurale netwerk weet wat er getraind moet worden, moeten de juiste kolommen uit de dataset geselecteerd worden. In dit voorbeeld gebruiken we als input de kolommen 'Setpoint (m)', 'Hoogte (m)', 'Fout', 'Fout_Integratie', 'Fout_Afgeleide' en als output de kolom 'PWM'. Dit kan als volgt in Python:

```
# Selecteer de input features en de target output
features = CSV_DATA[['Setpoint (m)', 'Hoogte (m)', 'Fout', 'Fout_Integratie', 'Fout_Afgeleide']].to_numpy()
target = CSV_DATA['PWM'].to_numpy()
```

Hiermee worden de input- en outputdata klaargezet voor het trainen van het neurale netwerk. Het is aan te raden om de data te normaliseren, zodat alle invoervariabelen binnen hetzelfde bereik vallen. Dit helpt het neurale netwerk om sneller en stabieler te trainen.

```
from sklearn.preprocessing import MinMaxScaler

# Normaliseer alleen de features (inputdata)
feature_scaler = MinMaxScaler()
features_norm = feature_scaler.fit_transform(features)

# Indien gewenst kun je ook de target normaliseren (optioneel)
target_scaler = MinMaxScaler()
target_norm = target_scaler.fit_transform(target.reshape(-1, 1))
```

Gebruik deze genormaliseerde data bij het trainen van je model. Vergeet niet om bij het evalueren of toepassen van het model dezelfde scalers te gebruiken om nieuwe data te transformeren. Nadat de data is genormaliseerd, moet deze worden

omgezet naar een formaat dat geschikt is voor het trainen van een neurale netwerk, met name wanneer je werkt met tijdreeksen of sequentiële data. In dit geval willen we het model leren voorspellen wat de PWM-uitgang moet zijn op basis van een reeks opeenvolgende meetwaarden. Hiervoor maken we gebruik van zogenaamde "vensters" (windows) over de data: voor elk trainingsvoorbeeld nemen we een aantal opeenvolgende tijdstappen als input (features), en voorspellen we de PWM-waarde van de volgende tijdstap (target). Stel dat je `n_input` als het aantal tijdstappen kiest dat je als input aan het model wilt geven. Dan kun je de volgende code gebruiken om de data in de juiste vorm te zetten:

```
X, y = [], []
for i in range(len(features_norm) - n_input):
    X.append(features_norm[i:i + n_input])
    y.append(target_norm[i + n_input]) # de PWM bij het volgende moment

X = np.array(X) # shape: (samples, n_input, aantal_features)
y = np.array(y)
```

- X bevat nu voor elk training voorbeeld een blok van `N_input` opeenvolgende tijdstappen met alle geselecteerde features.
- Y bevat de bijbehorend PWM-waarde die het model moet leren voorspellen.

Deze aanpak is vooral handig bij het trainen van modellen zoals recurrente neurale netwerken (RNNs) of convolutionele netwerken voor tijdreeksen, omdat het model zo leert om patronen in opeenvolgende data te herkennen. Pas de waarde van `n_input` aan op basis van hoeveel historie je het model wilt laten gebruiken bij het voorspellen.

Train-test splitting

Nadat de data in het juiste formaat is gezet, moet deze worden opgesplitst in trainingsdata en testdata. Dit is belangrijk om te kunnen beoordelen hoe goed het model presteert op nieuwe, ongeziene data. In dit voorbeeld wordt 20% van de data gebruikt als testset en 80% als training set. Je kunt handmatig bepalen vanaf welk punt in de dataset de testdata begint.

```
# Bepaal de lengte van de dataset
n = len(y)

# Bepaal het aantal samples voor de testset (20%)
test_len = int(n * 0.2)

# Kies het startpunt voor de testset, bijvoorbeeld index 3000
test_start = 3000
test_end = test_start + test_len

# Maak de train- en testsets
X_test, y_test = X[test_start:test_end], y[test_start:test_end]
X_train = np.concatenate((X[:test_start], X[test_end:]), axis=0)
y_train = np.concatenate((y[:test_start], y[test_end:]), axis=0)
```

- **X_train, Y_train:** data waarmee het model wordt getraind.
- **X_test, Y_test:** data waarmee het model wordt getest.

Let op: Zorg ervoor dat `test_start` en `test_end` binnen de grenzen van je dataset vallen. Je kunt deze waarden aanpassen afhankelijk van de grootte van je dataset en de gewenste verdeling.

Model bouwen

Nu gaan we het neurale netwerk opzetten dat de PWM-waarde voorspelt op basis van de genormaliseerde inputdata. We gebruiken hiervoor een sequentieel model met een 1D-convolutielaaag, gevolgd door een flatten-laag en twee dense (volledig verbonden) lagen. Dit model is geschikt voor het herkennen van patronen in tijdreeksen. Hieronder vind je de voorbeeldcode voor het bouwen van het model:

```
from keras.models import Sequential
from keras.layers import Conv1D, Flatten, Dense

model = Sequential([
    Conv1D(filters=8, kernel_size=2, activation='relu', input_shape=(n_input, X.shape[2])), # 1D-convolutielaaag
    Flatten(),
    Dense(8, activation='relu'), # Verborgen laag 1
    Dense(8, activation='relu'), # Verborgen laag 2
    Dense(1) # Uitgangslaag voor de PWM-waarde
])

model.summary()
```

- **Conv1D-laag:** herkent patronen in opeenvolgende tijdstappen van de inputdata.
- **Flatten-laag:** Zet de output van Conv1D-laag om naar een 1D-vector.
- **Dense-lagen:** Leren complexe relaties tussen de input en de gewenste output.
- **Outputlaag:** geeft de voorspelde PWM-waarde.

Zorg ervoor dat n_input overeenkomt met het aantal tijdstappen dat je als input gebruikt, en dat X.shape[2] gelijk is aan het aantal features per tijdstap. Met model.summary() krijg je een overzicht van de architectuur van het model, inclusief het type lagen, de outputvormen per laag en het aantal trainbare parameters. Dit helpt om te controleren of het model correct is opgebouwd en of de inputvormen kloppen. Zie foto hieronder.

Layer (type)	Output Shape	Param #
conv1d (Conv1D)	(None, 1, 8)	88
flatten (Flatten)	(None, 8)	0
dense (Dense)	(None, 8)	72
dense_1 (Dense)	(None, 8)	72
dense_2 (Dense)	(None, 1)	9

Total params: 241 (964.00 B)

Trainable params: 241 (964.00 B)

Non-trainable params: 0 (0.00 B)

Model compileren en trainen

Nu het model is opgebouwd, moet het worden gecompileerd en getraind. Compileren betekent dat je het model voorbereidt op het leerproces door een optimizer en een loss-functie te kiezen. In dit geval gebruiken we de Adam-optimizer en de Mean Squared Error (MSE) als loss-functie, wat gebruikelijk is bij regressieproblemen zoals het voorspellen van een PWM-waarde. Gebruik de volgende code om het model te compileren en te trainen:

```
# Compileer het model met de Adam-optimizer en MSE-loss
model.compile(optimizer='adam', loss='mse')

# Train het model op de trainingsdata, met validatie op de testdata
history = model.fit(
    X_train, y_train,
    epochs=5,           # Aantal keer dat het model de hele dataset doorloopt
    batch_size=32,       # Aantal samples per trainingsstap
    validation_data=(X_test, y_test)  # Gebruik testdata voor validatie tijdens training
)
```

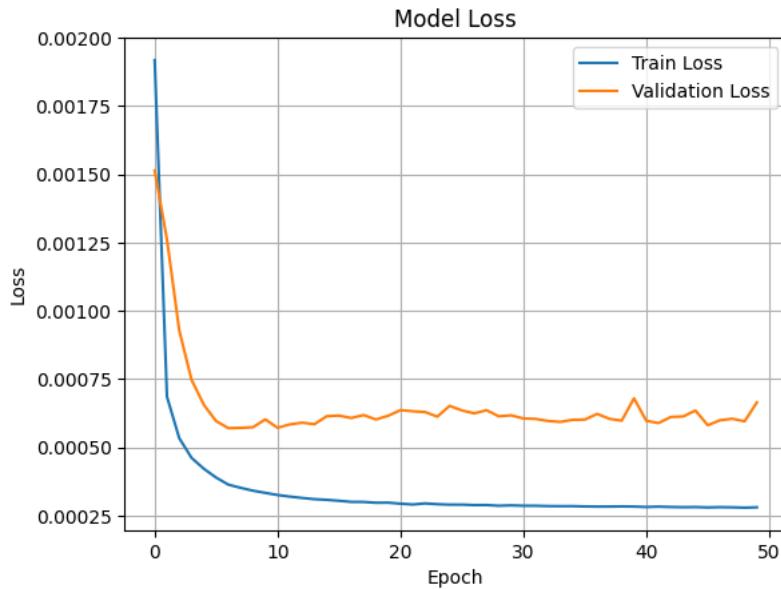
Tijdens het trainen houdt het model bij hoe de loss (fout) zich ontwikkelt op zowel de trainingsdata als de testdata (validatie). Dit wordt opgeslagen in het history-object. Om het leerproces inzichtelijk te maken, kun je de loss per epoch plotten:

```
import matplotlib.pyplot as plt

# Haal de loss-waarden op uit het history-object
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)

# Plot de trainings- en validatie-loss
plt.figure()
plt.plot(epochs, loss, label='Train Loss')
plt.plot(epochs, val_loss, label='Validation Loss')
plt.title('Model Loss per Epoch')
plt.xlabel('Epoch')
plt.ylabel('Loss (MSE)')
plt.legend()
plt.grid(True)
plt.show()
```

Hiermee krijg je een grafiek te zien van de loss tijdens het trainen. Een dalende loss betekent dat het model leert. Als de validatie-loss veel hoger is dan de trainings-loss, kan er sprake zijn van overfitting. Zie hieronder de grafiek.



Model evalueren

Nu het model getraind is, is het belangrijk om te controleren hoe goed het presteert op de testdata. Dit doe je door het model te valideren: je laat het voorspellingen doen op de test set en vergelijkt deze met de werkelijke waarden. Hieronder vind je code om het model te evalueren en de resultaten te visualiseren:

```
import numpy as np
import matplotlib.pyplot as plt

# Maak voorspellingen op de testset
predictions = model.predict(X_test).flatten()

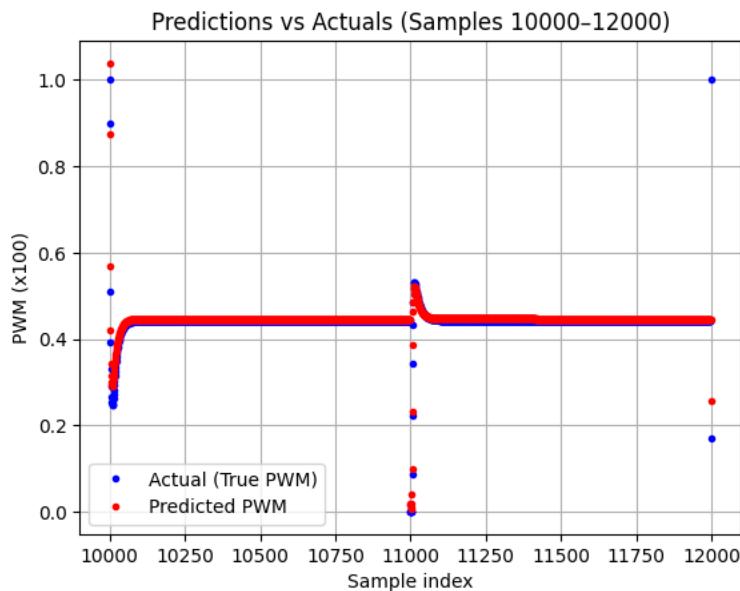
# Evalueer het model op de testdata
score = model.evaluate(X_test, y_test)
print(f"\n{model.metrics_names[0]} : {score}")

print(f"y_test heeft {len(y_test)} samples (indexbereik: 0 t/m {len(y_test) - 1})")

# Stel het bereik in voor de plot
start = 10000
end = min(12000, len(y_test))

# Plot de voorspellingen versus de werkelijke waarden
plt.figure()
plt.title(f"Predictions vs Actuals (Samples {start}-{end})")
plt.plot(range(start, end), y_test[start:end], 'b.', label='Actual (True PWM)')
plt.plot(range(start, end), predictions[start:end], 'r.', label='Predicted PWM')
plt.xlabel("Sample index")
plt.ylabel("PWM (x100)")
plt.legend()
plt.grid(True)
plt.show()
```

Met deze grafiek kun je visueel beoordelen hoe goed de voorspellingen van het model overeenkomen met de werkelijke PWM-waarden. Grote afwijkingen kunnen wijzen op onder- of over fitting, of op een model dat nog verder geoptimaliseerd moet worden. zie hieronder de grafiek.



Model opslaan en exporteren

Nadat het model is getraind en geëvalueerd, kun je het opslaan en exporteren zodat het gebruikt kan worden op een embedded systeem, bijvoorbeeld met TensorFlow Lite (TFLite). Dit maakt het mogelijk om het getrainde model te integreren in hardware zoals microcontrollers of andere edge devices. Hieronder vind je een voorbeeld van hoe je het model exporteert naar het TensorFlow SavedModel-formaat en vervolgens converteert naar het TFLite-formaat:

```
# Sla het model op in het TensorFlow SavedModel-formaat
model.export("model") # Exporteer als TensorFlow SavedModel voor TFLite

# Converteer het opgeslagen model naar TensorFlow Lite
converter = tf.lite.TFLiteConverter.from_saved_model("model")
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS]

tflite_model = converter.convert()
```

4.5.2: EIQ MODEL

De via NXP eiQ ontwikkelde AI beloofde een nauwkeurigheid van bijna 100% op een specifiek model. Dit model is als bibliotheek geëxporteerd en geïmporteerd in het eerder behandelde project tijdens sprint 2. De werking is als volgt:

```
timer_expired = false;
    hoogte = distance / 1000;
    setpointfloat = setpoint / 10000;
    fout = setpointfloat - hoogte;
    | fout_integratie += fout * Ts;
    | fout_afgeleide = (fout - vorige_fout) / Ts;
    | vorige_fout = fout;
```

Figuur 22: Variabelen genereren

De variabelen *hoogte*, *setpointfloat*, *fout*, *fout_integratie* en *fout_afgeleide* worden op basis van de samptijd, het setpoint en de hoogte berekend. Deze variabelen zijn benodigt voor de werking van de AI en zijn allemaal van het type *float*.

```
float channels[5] = { setpointfloat, hoogte, fout, fout_integratie, fout_afgeleide };

tss_reg_predict(channels, target);

int AIPWM = (int)target[0]*100;
```

Figuur 23: Tss Predict

Vervolgens worden alle variabelen in een array geplaatst. Met behulp van de functie *tss_reg_predict* uit de library wordt daarna de PWM-waarde voorspeld. Hierbij wordt het array *channels* als invoer gebruikt, en het resultaat wordt opgeslagen in het array *target*. Dit resultaat wordt daarna omgezet naar een integerwaarde in de gewenste grootheid en opgeslagen in *AIPWM*.

```
if (AIPWM < 0) AIPWM = 0;
if (AIPWM > 100) AIPWM = 100;

pwm = AIPWM;
sendThreeIntsUART(filtered measurement, (pwm*10), setpoint);
```

Figuur 24: Randvoorwaarden en UART

Daarna worden randvoorwaarden toegepast op het PWM-signal, waarna dit signaal via UART naar de seriële monitor wordt verzonden.

In deze sprint werkte het gehele AI-model echter nog niet, doordat de geïmporteerde library niet correct gekoppeld was.

4.6: SPRINT 6 – AFRONDING EN OPLEVERING

Sprint 6 stond in het teken van afronden. In deze fase zijn de laatste technische issues aangepakt en is de volledige documentatie opgesteld. Hoewel niet alle onderdelen volledig succesvol zijn afgerond — sommige implementatieproblemen met de AI-modellen konden binnen de beschikbare tijd niet volledig opgelost worden — zijn alle bevindingen, uitdagingen en (tussen)resultaten zorgvuldig vastgelegd. Zowel de werkende als de minder goed werkende onderdelen zijn geëvalueerd en beschreven. Op deze manier biedt het einddocument een volledig overzicht van het uitgevoerde onderzoek, inclusief de opgedane inzichten en leerervaringen, die waardevol zijn voor vervolgonderzoek en toekomstige projecten.

5: ONDERZOEK

Tijdens dit project hebben we twee deelonderzoeken uitgevoerd, elk met een eigen focusgebied binnen de toepassing van onze microcontroller. In het eerste onderzoek hebben we onderzocht welke mogelijkheden beschikbaar zijn binnen de toolkit die door NXP wordt geleverd. Hierbij is gekeken naar de functionaliteit, de ontwikkelomgeving en de ondersteuning voor embedded ontwikkeling specifiek voor onze controller.

In het tweede onderzoek wordt onderzocht of een neurale netwerk het gedrag van een PID-regelaar kan nabootsen in een systeem waarbij een pingpongbal op een ingestelde hoogte wordt gehouden. De huidige regeling werkt met een PID-regelaar, maar het afstellen hiervan is tijdverdurend en gevoelig voor omgevingsfactoren. Door een machine learning-model te trainen met data van het systeem, wordt onderzocht of een alternatief regelalgoritme mogelijk is. Er wordt niet ingegaan op de implementatie van het netwerk in een embedded omgeving, maar er wordt wel rekening gehouden met de beperkingen van de NXP FRDM-MCXN947 microcontroller, waarop het model uiteindelijk moet draaien. Het onderzoek richt zich op de technische haalbaarheid en de prestatievergelijking tussen het neurale netwerk en de traditionele PID-regelaar.

In dit hoofdstuk worden beide onderzoeken afzonderlijk besproken, inclusief de werkwijze, bevindingen en conclusies.

5.1: NXP TOOLKIT (eIQ)

De eIQ Time Series Studio van NXP is in dit project uitgebreid onderzocht en toegepast voor de ontwikkeling van een embedded AI-oplossing. Tijdens het werken met de tool werd duidelijk dat het direct verzamelen van data via UART niet altijd vlekkeloos verloopt. Desondanks bood de mogelijkheid om externe datasets te importeren in CSV-formaat een praktische en effectieve oplossing, waarmee het ontwikkelproces succesvol kon worden voortgezet.

Binnen de Time Series Studio bleken vooral de analysetools, zoals de *Data Intelligence*-functie, zeer waardevol. Hiermee kon de kwaliteit van de dataset automatisch worden gecontroleerd en geoptimaliseerd, wat de basis legde voor een goed getraind AI-model. Vervolgens kon met behulp van de ingebouwde *Deployment*-functionaliteit eenvoudig een kant-en-klare library file (libtss) gegenereerd worden. Deze gegenereerde bibliotheek maakt het mogelijk om het getrainde model snel te integreren in embedded toepassingen zonder ingewikkelde handmatige stappen.

Samenvattend biedt de eIQ-toolkit, en met name de Time Series Studio, een toegankelijke en goed ondersteunde ontwikkelomgeving voor het bouwen van AI-modellen op embedded hardware. De combinatie van automatische data-analyse, eenvoudige modeltraining en een soepele deployment maakt deze toolset goed geschikt voor embedded AI-projecten waarbij tijdsdata centraal staat.

Deze evaluatie is gebaseerd op het onderzoek *Onderzoeksverslag – Mogelijkheden ontwikkelen AI met NXP tools*, uitgevoerd door Piet Poot en Yered Scheffer in opdracht van EAS Datalab HR. Dit onderzoek is te vinden in de bijlage.

5.2: ONDERZOEK MITCHEL EN THOMAS (IS HET MOGELIJK OM EEN PID CONTROLLER TE VERVANGEN DOOR EEN AI?)

In dit onderzoek wordt onderzocht of een neurale netwerk het gedrag van een klassieke PID-regelaar kan nabootsen of zelfs verbeteren binnen een systeem waarin een pingpongbal op een constante hoogte wordt gehouden door middel van een ventilator. De huidige opstelling op de Hogeschool Rotterdam gebruikt een microcontroller en een PID-regelaar om de hoogte van de bal te regelen. Het afstemmen van deze regelaar is echter gevoelig voor omgevingsveranderingen en vereist handmatige tuning. Met de opkomst van machine learning rijst de vraag of een zelflerend model betere prestaties kan leveren. In dit project is een neurale netwerk ontworpen, getraind en getest in een gesimuleerde omgeving. Het model is geëvalueerd op nauwkeurigheid, stabiliteit en generaliseerbaarheid, en vervolgens vergeleken met de klassieke PID-regelaar. De resultaten tonen aan dat het neurale netwerk het gedrag van de PID-regelaar goed kan benaderen en in sommige gevallen zelfs verbeterde prestaties levert, mits correct getraind. Hoewel de uiteindelijke implementatie op embedded hardware buiten de scope van dit onderzoek valt, is bij de modelkeuze wel rekening gehouden met beperkingen van embedded systemen. Dit onderzoek biedt waardevolle inzichten in het toepassen van kunstmatige intelligentie voor regeltechniek in embedded omgevingen en dient als opstap voor verdere integratie van slimme regelaars in praktische toepassingen.

6: IMPLEMENTATIE

Uiteindelijk zijn er twee verschillende implementaties ontwikkeld: een zelfgebouwd model en een model dat is opgezet met behulp van de eIQ Tool-kit. Beide implementaties worden hieronder verder toegelicht.

6.1: ZELFONTWIKKELDE MODEL

Dit zelfontwikkeld model wordt in hoofdstuk 4.5.1: Zelfontwikkelde model besproken met onderbouwing van de code en de resultaten worden besproken in het onderzoek in 9.2: Bijlage 2 – Onderzoeksverslag – Is het mogelijk om een PID controller te vervangen door een AI?.

6.2: EIQ MODEL

6.2.1: DATASET EN MODELSELECTIE IN DE EIQ TOOLKIT

Voor de NXP eIQ Toolkit is er een dataset gegenereerd op basis van een simulatie die in het vorige hoofdstuk is toegelicht. Deze dataset bestaat uit 1.000.000 samples en bevat in totaal 6 kolommen, elk met een specifieke betekenis binnen de context van het regelsysteem voor de hoogtecontrole van een pingpongbal:

- **Setpoint:** de gewenste (doel)hoogte van de pingpongbal.
- **Hoogte:** de huidige gemeten hoogte van de pingpongbal.
- **Fout (Error):** het verschil tussen de gewenste hoogte (setpoint) en de gemeten hoogte.
- **Foutintegratie:** de som van alle voorgaande fouten; dit geeft aan of de fout over tijd cumulatief toeneemt of afneemt.
- **Foutafgeleide:** de verandering van de fout ten opzichte van de tijd; dit helpt om snelle variaties in fout te detecteren.
- **PWM (Pulse Width Modulation):** de aansturingswaarde van de actuator (bijvoorbeeld een ventilator of motor), oftewel het doelwit of de target output van het model.

De eIQ-toolkit verdeelt de dataset automatisch in 80% training en 20% validatie, wat gebruikelijk is in machine learning om overfitting te voorkomen en de generalisatiecapaciteit van het model te testen.

6.2.2: MODELSELECTIE: REGRESSIEMODELEN

Binnen de eIQ Toolkit is er gekozen voor de modelcategorie 'regressie', omdat het doel is om een continue waarde (PWM) te voorspellen op basis van invoerwaarden. De volgende regressiemodellen zijn getraind en geëvalueerd:

1. Ridge Regression

Een lineair regressiemodel met L2-regularisatie. Ridge probeert overfitting tegen te gaan door de grootte van de gewichten te beperken.

2. XGBoost (Extreme Gradient Boosting)

Een krachtig model gebaseerd op beslissingsbomen dat gebruik maakt van graduele verbetering van fouten in eerdere modellen (boosting). Geschikt voor complexe datasets met veel interacties.

3. CatBoost

Vergelijkbaar met XGBoost, maar geoptimaliseerd voor categorische data en schnellere training. Kan goed omgaan met ongeschaalde inputdata.

4. LightGBM (Light Gradient Boosting Machine)

Een lichtgewicht, snelle implementatie van gradient boosting die schaalbaar is naar grote datasets.

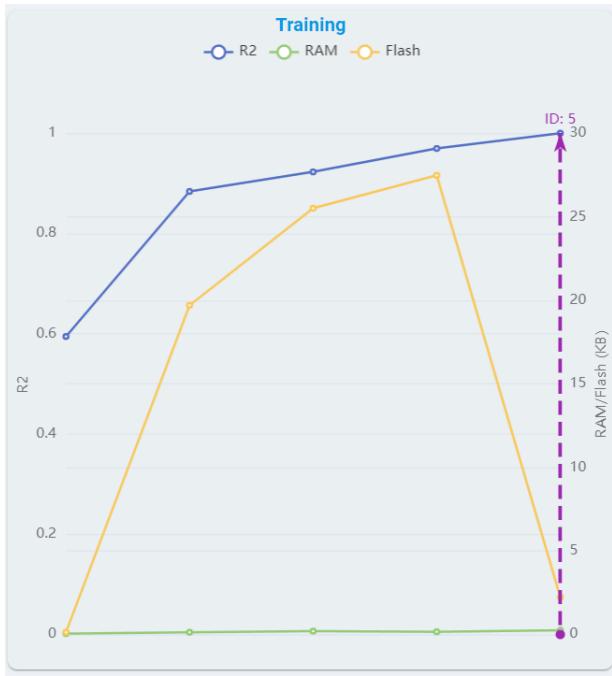
5. MLP (Multilayer Perceptron)

Een type **kunstmatig neuraal netwerk** dat bestaat uit meerdere lagen van neuronen: een inputlaag, één of meerdere verborgen lagen en een outputlaag. MLP's zijn krachtig in het modelleren van niet-lineaire relaties tussen invoer en uitvoer, wat erg belangrijk is bij dynamische regelsystemen zoals deze toepassing.

6.2.3: EVALUATIE VAN DE MODELLEN

De modellen zijn geëvalueerd op basis van de R²-score (coëfficiënt van determinatie). Deze score geeft aan hoe goed het model de variantie in de data verklaart:

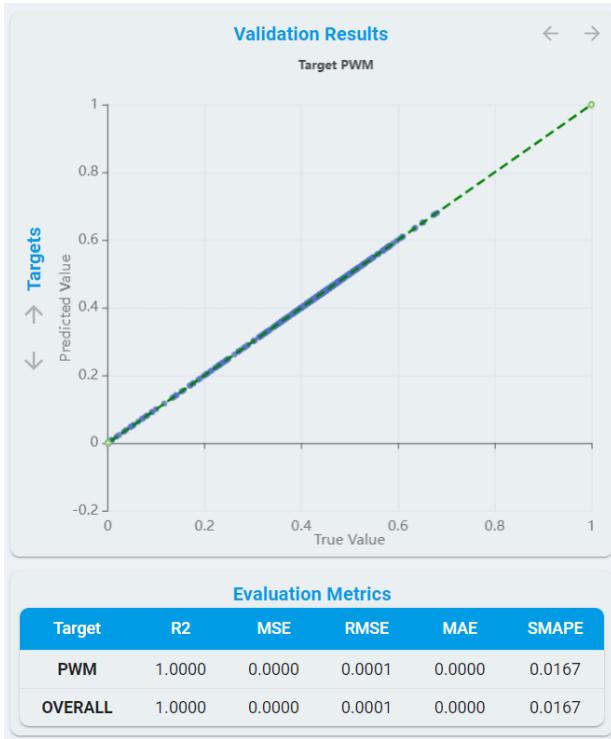
- Een R²-score van 1.0 betekent dat het model perfecte voorspellingen maakt.
- Een score van 0.0 betekent dat het model niet beter presteert dan een simpele gemiddelde-voorspelling.
- Een negatieve score duidt op een model dat slechter presteert dan een naïef model.



Figuur 25: Training

De behaalde resultaten waren als volgt:

Model	R ² -score
MLP	1.000 (100%)
CatBoost	~0.90
LightGBM	~0.88
XGBoost	~0.85
Ridge	~0.60



Figuur 26: MLP Model

Gezien de uitstekende prestaties van het MLP-model, is ervoor gekozen dit model verder te gebruiken in de volgende stappen van het project.

6.2.4: INTERGRATIE

In de Deployment tab kon er gekozen worden uit verschillende instellingen.

- Je model die je wilt gebruiken: MLP
- Welke CPU core je hebt: CORTEX-M33
- Een libray of een project voor de MCUXpresso: wij hebben gekozen voor een makkelijkere intergratie voor ons systeem en dat kwam uit op een library
- Je toolchain selectie: dit was voor onze board een GCC

Ook waren er extra instellingen waar niet voor gekozen is om aan te zitten, want dat waren instellingen die vooral specifieke onderdelen sneller of efficienter kunnen laten werken, zoals minder RAM gebruik of andere soorten Floats. Omdat er nog geen diepere kennis over is zijn we voor de standaard instellingen gegaan.

```

Hello World Example

#include "TimeSeries.h"

float data_input[TSS_INPUT_DATA_LEN * TSS_INPUT_DATA_DIM];

void sample_data(float data_buffer[])
{
    /* Collect data and put them into the buffer */
}

int main(void)
{
    tss_status status;
    float targets[TSS_TARGET_NUMBER];

    status = tss_reg_init();
    if (status != TSS_SUCCESS)
    {
        /* Handle the initialization failure cases */
    }

    while (1)
    {
        sample_data(data_input);
        status = tss_reg_predict(data_input, targets);
        if (status != TSS_SUCCESS)
        {
            /* Handle the prediction failure cases */
        }

        /* Handle the prediction result */
    }

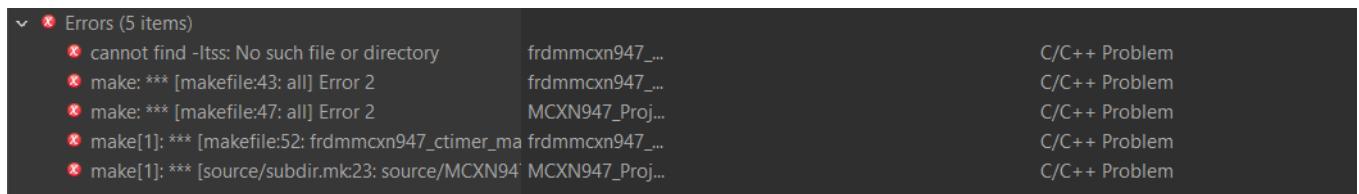
    return 0;
}

```

Figuur 27: Hello world

Dit leverde een *HelloWorld*-voorbeeld op, waarin functies uit de library worden gebruikt om de AI aan te roepen en de voor-spelde waarde uit te lezen. De library is vervolgens geëxporteerd en geïmporteerd in de hardwarecode. Deze import is terug te vinden in het hoofdstuk *Sprints*.

Helaas was de code niet direct testbaar, doordat er een foutmelding optrad waarbij de library niet gevonden werd.



Figuur 28: Errors

Tijdens het compileren van de hardwarecode traden meerdere fouten op, zoals te zien is in onderstaande foutmelding:

- cannot find -ltss: No such file or directory
- Vervolgens meerdere makefile fouten (Error 2)

De foutmelding cannot find -ltss betekent dat de compiler/linker probeert een externe library met de naam tss te linken, maar deze library is niet gevonden op het systeem. Omdat deze externe library ontbreekt, faalt de linker en worden de volgende regels in het makefile afgebroken met foutcode 2. De andere foutmeldingen zijn hier een gevolg van: het buildproces stopt zodra de linker deze ontbrekende library niet kan vinden.

Waarschijnlijk is de vereiste library in het project niet goed opgenomen in de build-omgeving. Mogelijke oorzaken kunnen zijn:

- De library is niet geïnstalleerd.
- De library staat niet op de juiste plek.

- De linker zoekt niet op de juiste zoekpaden.

Door deze fout kan het project niet succesvol gecompileerd worden en is testen van de hardwarecode niet mogelijk.

Dit probleem is tot op heden nog niet opgelost.

7: CONCLUSIE

In dit project is met succes onderzocht hoe een neurale netwerk het gedrag van een klassieke PID-regelaar kan nabootsen om de hoogte van een pingpongbal in een luchtstroom te reguleren. Hierbij zijn alle stappen doorlopen van modelontwikkeling tot gedeeltelijke implementatie op embedded hardware.

Het project begon met de nodige theoretische verdieping in neurale netwerken en PID-regeling. Vanuit eenvoudige AI-voorbeelden met TensorFlow is kennis opgebouwd die uiteindelijk werd toegepast in het trainen van een neurale netwerk op een zelfgesimuleerde dataset van een PID-regelaar. Dankzij deze simulatie konden realistische trainingsdata worden geleverd, wat essentieel was voor het correct aanleren van het netwerk.

Een proof-of-concept is succesvol uitgevoerd waarbij een eenvoudig neurale netwerk werd getraind om een sinusgolf te voorspellen. Dit model is vervolgens omgezet naar TensorFlow Lite en geïntegreerd in een C-headerbestand voor gebruik op de FRDM-MCXN947 microcontroller. Hoewel de uiteindelijke implementatie van het volledige AI-regelmodel op de hardware nog niet volledig is afgerond, is de technische basis gelegd, inclusief modelconversie, datastructuur, en microcontrollerintegratie via TensorFlow Lite Micro.

Kortom, het project toont aan dat het vervangen van een PID-regelaar door een neurale netwerk technisch haalbaar is. De opgedane kennis en ervaring vormen een waardevolle basis voor verdere ontwikkeling en fine-tuning van AI-gebaseerde regelsystemen in embedded toepassingen.

8: BIBLIOGRAFIE

- [1] GeeksforGeeks, „GeeksforGeeks,” 2024. [Online]. Available: <https://www.geeksforgeeks.org/recurrent-neural-networks-rnn/>.

Onderzoeksverslag – Mogelijkheden ontwikkelen AI met NXP tools

in opdracht van EAS Datalab HR

Yered Scheffer; Piet Poot

Abstract — In dit onderzoek is de eIQ Toolkit van NXP geëvalueerd met als doel te bepalen in hoeverre deze geschikt is voor het ontwikkelen van machine learning-toepassingen op embedded microcontrollers. De focus lag op het genereren van datasets, het optimaliseren en trainen van modellen, de integratie met de MCUXpresso IDE, en de bruikbaarheid van de bijgeleverde documentatie en voorbeeldprojecten. De resultaten bieden inzicht in de toepasbaarheid van eIQ voor embedded AI en geven een overzicht van de ontwikkelervaring binnen de NXP-omgeving.



INLEIDING

In dit onderzoek is geëxperimenteerd met de **eIQ Toolkit** van NXP, ontwikkeld voor gebruik met de **FRDM-MCXN947**-ontwikkelplaat. Het doel was om te onderzoeken in hoeverre deze software geschikt is voor het ontwikkelen van een AI-toepassing op dit specifieke bord. In deze rapportage worden de verschillende onderzochte aspecten, waaronder de ontwikkeling, training en integratie van AI-modellen, stapsgewijs toegelicht.

BASISWERKING SOFTWARE

De benodigde software is eenvoudig te downloaden via de website van NXP. Volgens de beschikbare documentatie biedt de eIQ Toolkit functionaliteit voor het trainen van AI-modellen, het genereren van datasets, het optimaliseren van modellen, en het direct implementeren van deze modellen op embedded microcontrollers.

De basisfunctionaliteit van de toolkit is primair gericht op beeldherkenning. Omdat dit onderzoek zich richtte op een meer geavanceerde vorm van AI-implementatie, is verder gekeken naar de mogelijkheden binnen de Time Series Studio, een geïntegreerd onderdeel van de eIQ Toolkit. Deze module

stelt ontwikkelaars in staat om complexere AI-modellen te bouwen op basis van tijdreeksdata, wat relevant is voor toepassingen zoals sensorgebaseerde voorspellingen of regelstrategieën. Om die reden is het vervolg van dit onderzoek specifiek gericht op deze module.

De Time Series Studio bevat eigen documentatie die uitsluitend beschikbaar is binnen de tool zelf. Deze documentatie is niet extern toegankelijk via internet, wat de toegankelijkheid voor voorbereiding en oriëntatie enigszins beperkt..

GENEREREN DATASET

Binnen de Time Series Studio is het mogelijk om datasets te genereren door een extern systeem via UART te koppelen aan de software. In dit onderzoek is gebruikgemaakt van een opstelling bestaande uit een pingpongbal in een verticale buis, waarbij de positie van de bal wordt gereguleerd met een ventilator en gemeten met een ultrasoonsensor. Deze opstelling werd ingezet om de werking van de toolkit te testen. Hoewel de verbinding met de software succesvol tot stand werd gebracht, bleek de gegenereerde data niet bruikbaar. Naar aanleiding hiervan is contact opgenomen met NXP, maar tot op heden is hierop geen reactie ontvangen.

Voor het genereren van een dataset moet eerst worden vastgesteld welke variabelen dienen als input en welke als output van het AI-model. In de Time Series Studio worden deze respectievelijk aangeduid als channels (inputs) en targets (outputs).

Hoewel het precieze gebruik van channels en targets licht kan verschillen per modeltype, geldt in grote lijnen de volgende interpretatie:

- Channels: vertegenwoordigen de verschillende soorten invoerwaarden of meerdere instanties van dezelfde invoer in verschillende omstandigheden.
- Targets: geven de verwachte outputwaarden weer, die het model tijdens de training leert te voorspellen.

Bij het opstellen van de dataset is het belangrijk om ook de targets op te nemen in de channels-sectie, omdat deze bij de meeste modeltypen als onderdeel van de invoer worden beschouwd. Dit betekent in de praktijk:
Channels = Inputs + Targets

Naast het direct genereren van datasets via een live UART-verbinding, biedt Time Series Studio ook de mogelijkheid om datasets in CSV-formaat te importeren. In dit onderzoek is gekozen voor deze methode. De dataset werd extern opgesteld, gevalideerd en vervolgens succesvol geïmporteerd in de tool. Deze aanpak bleek wél effectief en heeft bruikbare resultaten opgeleverd. Om die reden is deze werkwijze als uitgangspunt genomen voor het verdere verloop van het onderzoek.

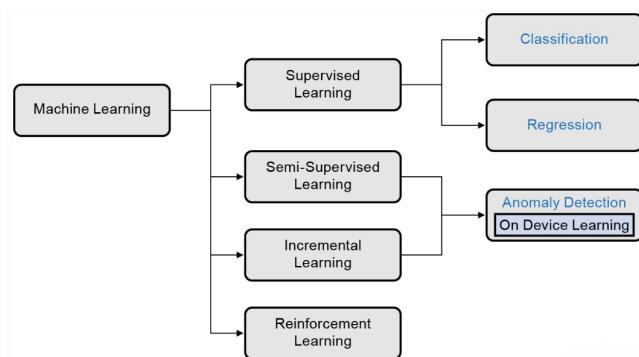
Na selectie van een dataset wordt deze automatisch gesplitst in een verhouding van 80/20%, waarbij 80% van de gegevens wordt gebruikt voor training en 20% voor validatie en testen.

Tot slot is het van essentieel belang om de juiste documentatie te raadplegen bij het opstellen van een dataset. Met name het hoofdstuk "Time Series Data" in de bijgeleverde handleiding biedt cruciale informatie over de vereiste structuur van datasets per modeltype. Hierin wordt per model uitgelegd welk formaat van de invoerdata verwacht wordt en hoe deze correct moet worden opgebouwd..

MODEL TRAINING

Voordat een dataset gebruikt kan worden voor het trainen van een AI-model binnen Time Series Studio, dient de gebruiker een aantal parameters in te stellen en het type model te kiezen. De tool biedt drie hoofdtypen modellen:

- **Anomaly Detection:** Gericht op het identificeren van datapunten die significant afwijken van het normale patroon. Voor het trainen van een dergelijk model is een dataset vereist die zowel normale als afwijkende gegevens bevat. In tegenstelling tot classificatie is anomaliedetectie vaak gebaseerd op unsupervised learning, waarbij volledige labeling van data niet noodzakelijk is.
- **Classification:** Hierbij worden gegevens ingedeeld in vooraf gedefinieerde categorieën. Voor classificatie zijn minstens twee klassen nodig. Het model wordt getraind op een gelabelde dataset waarin elk datapunt behoort tot een specifieke categorie.
- **Regression:** Regressiemodellen voorspellen continue waarden op basis van één of meerdere invoervariabelen. Dit modeltype is geschikt voor het modelleren van relaties en het doen van voorspellingen op basis van numerieke data.



[2] Time Series Studio Documentation

Projectconfiguratie

Na het kiezen van het modeltype wordt er een nieuw project aangemaakt in Time Series Studio. Hierbij moet de gebruiker:

- Het gewenste modeltype selecteren.
- De doelsysteemhardware kiezen (in dit onderzoek: FRDM-MCXN947).
- Het aantal channels (inputs + targets) en targets (outputs) opgeven.

Voor dit onderzoek is gekozen voor een project met 5 channels en 1 target. De bijbehorende dataset bevat 4 invoervariabelen (bijv. setpoint, gemeten hoogte, vorige PWM, etc.) en 1 target die het gedrag van een PID-regelaar simuleert..

Na het aanmaken van het project zijn vier hoofdonderdelen beschikbaar:

1. **Dataset**
2. **Training**
3. **Emulation**
4. **Deployment**

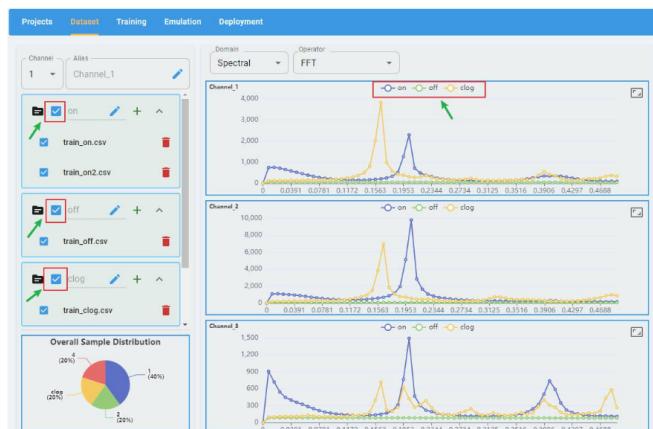
In dit hoofdstuk wordt de focus gelegd op de eerste twee onderdelen.

Datasetbeheer

De eerste stap is het uploaden van de dataset. Na het uploaden moet de structuur van de dataset worden aangegeven:

- Welke kolommen behoren tot de channels.
- Welke kolom fungereert als target.
- Hoe de data verdeeld is over de rijen (tijdsvolgorde).

De tool biedt daarnaast een visuele validatie van de dataset, waarmee gecontroleerd kan worden of de data correct is ingelezen en logisch verdeeld is.



[2] Time Series Studio Documentation

Task Type	Model Name	Full Name	Python Code Source	Python Code License	Deployment C	Code Source
Anomaly Detection (On-Device Learn)	ZSM	Z-Score Model	NXP	NXP	NXP	NXP
	IPCA	Incremental Principal Component Analysis	sckit-learn	BSD-3-Clause	NXP	NXP
	IMHL	Incremental Mahalanobis Distance Based Model	NXP	NXP	NXP	NXP
Anomaly Detection (No On-Device Learn)	IKM	Incremental KMeans	NXP	NXP	NXP	NXP
	ZSM	Z-Score Model	NXP	NXP	NXP	NXP
	OCSVM	One Class Support Vector Machine	sckit-learn	BSD-3-Clause	NXP	NXP
	GMM	Gaussian Mixture Model	sckit-learn	BSD-3-Clause	NXP	NXP
Classification	PCA	Principal Component Analysis	sckit-learn	BSD-3-Clause	NXP	NXP
	MHL	Mahalanobis Distance Based Model	NXP	NXP	NXP	NXP
	MLP	Multi Layer Perceptron	sckit-learn	BSD-3-Clause	NXP	NXP
	SVM	Support Vector Machine	sckit-learn	BSD-3-Clause	NXP	NXP
	LR	Random Forest	sckit-learn	BSD-3-Clause	NXP	NXP
Regression	XGBoost	Extreme Gradient Boosting	XGBoost	Apache 2.0	NXP	NXP
	CatBoost	Categorical Boosting	CatBoost	Apache 2.0	NXP	NXP
	LGBM	Light Gradient Boosting Machine	LightGBM	MIT	NXP	NXP
	Ridge	Categorical Boosting	CatBoost	Apache 2.0	NXP	NXP
	RIDGE	Ridge regression	sckit-learn	BSD-3-Clause	NXP	NXP

[2] Time Series Studio Documentation

Modeltraining

Zodra de dataset correct is geconfigureerd, kan het model getraind worden. Bij het trainen van een regressiemodel worden de volgende algoritmen standaard ingezet:

- RFR (Random Forest Regressor)
- MLP (Multilayer Perceptron)
- XGBoost (Extreme Gradient Boosting)
- LGBM (Light Gradient Boosting Machine)
- Ridge Regression

Tijdens de training wordt de dataset automatisch gesplitst in een 80/20-verhouding: 80% van de data wordt gebruikt voor training, en 20% voor validatie. In het Training-scherm van de tool zijn vervolgens de resultaten zichtbaar van elk model op basis van deze 20%.



[2] Time Series Studio Documentation

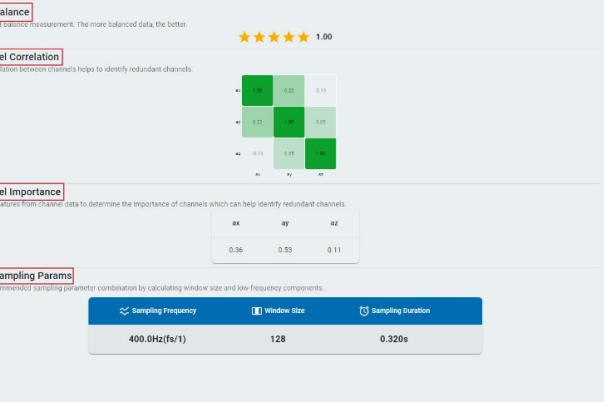
De belangrijkste onderdelen van het trainingsproces zijn:

- Training: Toont de prestaties van elk model, inclusief verbruik van RAM en Flash-geheugen, en een grafische vergelijking van de nauwkeurigheid.
- Benchmark: Geeft inzicht in de benodigde tijd voor training, evenals het geheugengebruik op de doelsysteemhardware.
- Validation Results: Bevat de resultaten van de validatiedata; idealiter ligt de voorspelling zo dicht mogelijk bij de werkelijke (target)waarden.
- Evaluation Metrics: Geeft aanvullende prestatie-indicatoren (zoals RMSE, MAE, R²-score), afhankelijk van het gekozen modeltype.

Op basis van deze resultaten wordt het best presterende model geselecteerd voor verdere stappen binnen de tool, zoals emulatie en deployment.

MODEL OPTIMALISATIE

Om het model goed te kunnen trainen, is het belangrijk dat de data van goede kwaliteit is. In dit project is hiervoor de *Data Intelligence*-functie van Time Series Studio gebruikt. Dit hulpmiddel controleert automatisch of de dataset geschikt is voor training en geeft suggesties voor verbeteringen.



[2] Time Series Studio Documentation

Tijdens deze analyse zijn de volgende onderdelen bekijken:

Balans van de data

Er is gecontroleerd of alle klassen ongeveer evenveel trainingsdata hebben. Een goede balans is belangrijk, zodat het model niet te veel leert van één bepaalde klasse. In dit geval was de data goed verdeeld over alle klassen.

Samenhang tussen kanalen:

Er is gekeken of de verschillende datakanalen veel op elkaar lijken (correlatie). Als kanalen sterk op elkaar lijken, kan dat onnodige herhaling zijn. In deze dataset waren de kanalen voldoende verschillend.

Belang van de kanalen

Voor elke meetwaarde is bepaald hoe belangrijk deze is voor het herkennen van de klassen. Minder belangrijke kanalen zouden eventueel weggeleggen kunnen worden om het model eenvoudiger te maken.

Advies voor de meetinstellingen

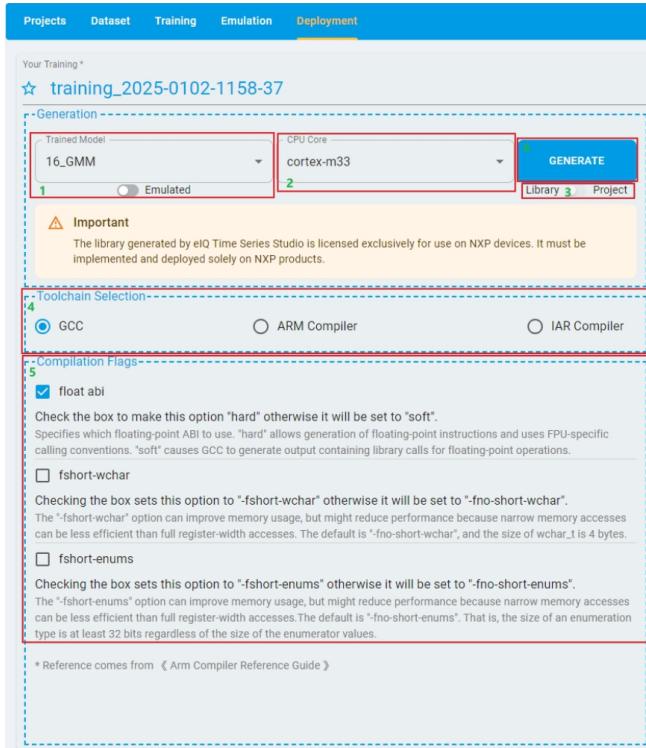
Er is een advies gegeven voor de beste meetfrequentie en venstergrootte (window size). Door bijvoorbeeld de meetfrequentie iets te verlagen, worden onnodige details (zoals ruis) weggefilterd. Hierdoor wordt het trainen sneller en blijft de belangrijke informatie behouden.

De aangepaste dataset is daarna opgeslagen en gebruikt om het model verder te trainen. Door deze automatische controle is de kans op fouten kleiner en verloopt het optimaliseren van het model sneller.

Deze functie binnen de toolkit was zeer bruikbaar en werkte probleemloos.

IMPLEMENTATIE

Na het trainen en testen van het model is het model geïmplementeerd met behulp van de *Deployment*-functionaliteit binnen de Time Series Studio. Deze functie maakt het mogelijk om het getrainde model om te zetten naar een kant-en-klare algoritme-bibliotheek die geschikt is voor embedded toepassingen.



[2] Time Series Studio Documentation

Binnen de deployment-module hebben we eerst ons beste model geselecteerd dat eerder tijdens de emulatie was gevalideerd. Vervolgens is gekozen voor het genereren van een library file (libtss), in plaats van een volledig project. Deze library bevat alle benodigde algoritmes in gecompileerde vorm, inclusief:

- de bibliotheek zelf (*libtss.a*),
- het headerbestand (*TimeSeries.h*) voor integratie in de software,
- metadata met informatie over het model en de hardware,
- en een licentiebestand.

Het grote voordeel van deze aanpak is dat de gegenereerde library eenvoudig geïntegreerd kan worden in de bestaande AI-opstelling. Door de standaard gegenereerde interface kunnen ontwikkelaars de bibliotheek eenvoudig koppelen aan hun eigen applicatiecode.

Voor dit onderzoek is alleen het proces tot en met het genereren van de library file van belang. De verdere implementatie op de daadwerkelijke hardware-opstelling valt buiten de scope van dit onderzoek.

Deze deployment-functionaliteit binnen Time Series Studio werkte erg goed en heeft het implementatieproces sterk vereenvoudigd.

CONCLUSIE

In dit onderzoek is de eIQ Time Series Studio van NXP succesvol onderzocht en toegepast voor het ontwikkelen van een embedded AI-toepassing. Ondanks enkele beperkingen bij het direct verzamelen van data via UART, bood de mogelijkheid om externe datasets te importeren een effectieve oplossing om het onderzoek voort te zetten.

De tools binnen Time Series Studio, zoals de *Data Intelligence*-functie, bleken zeer bruikbaar voor het analyseren en optimaliseren van de dataset. Hierdoor kon het model goed getraind worden op kwalitatieve data. Daarnaast heeft de ingebouwde *Deployment*-functionaliteit het mogelijk gemaakt om zonder veel handmatige stappen een kant-en-klare library file (libtss) te genereren. Deze library is direct inzetbaar voor verdere implementatie in embedded hardware.

Al met al biedt de eIQ-toolkit, en in het bijzonder de Time Series Studio, een gebruiksvriendelijke en goed gedocumenteerde omgeving voor het ontwikkelen van AI-modellen op embedded systemen. De combinatie van data-analyse, modelltraining en eenvoudige deployment maakt het een geschikte oplossing voor vergelijkbare AI-projecten binnen embedded toepassingen.

REFERENTIELIJST

- [1] "NXP Semiconductors | Automotive, Security, IoT," [Nxp.com](https://www.nxp.com/), 2019. <https://www.nxp.com/>
- [2] NXP, "Time Series Studio Documentatie"

Is het mogelijk om een PID controller te vervangen door een AI?

Mitchel Reints - 1040953@hr.nl

Thomas van Ommeren - 1033239@hr.nl

Hogeschool Rotterdam - Elektrotechniek - PEE51

21 juni 2025

Samenvatting

In dit onderzoek wordt onderzocht of een neurale netwerk het gedrag van een klassieke PID-regelaar kan nabootsen of zelfs verbeteren binnen een systeem waarin een pingpongbal op een constante hoogte wordt gehouden door middel van een ventilator. De huidige opstelling op de Hogeschool Rotterdam gebruikt een microcontroller en een PID-regelaar om de hoogte van de bal te regelen. Het afstemmen van deze regelaar is echter gevoelig voor omgevingsveranderingen en vereist handmatige tuning. Met de opkomst van machine learning rijst de vraag of een zelflerend model betere prestaties kan leveren. In dit project is een neurale netwerk ontworpen, getraind en getest in een gesimuleerde omgeving. Het model is geëvalueerd op nauwkeurigheid, stabilité en generaliseerbaarheid, en vervolgens vergeleken met de klassieke PID-regelaar. De resultaten tonen aan dat het neurale netwerk het gedrag van de PID-regelaar goed kan benaderen en in sommige gevallen zelfs verbeterde prestaties levert, mits correct getraind. Hoewel de uiteindelijke implementatie op embedded hardware buiten de scope van dit onderzoek valt, is bij de modelkeuze wel rekening gehouden met beperkingen van embedded systemen. Dit onderzoek biedt waardevolle inzichten in het toepassen van kunstmatige intelligentie voor regeltechniek in embedded omgevingen en dient als opstap voor verdere integratie van slimme regelaars in praktische toepassingen.

1 Inleiding

Hogeschool Rotterdam beschikt over de cursus Digitale Systemen (DIS10) over een opstelling waarbij een pingpongbal op een ingestelde hoogte wordt gehouden. Deze opstelling bestaat uit een verticale buis met onderaan een ventilator. Door de luchtdruk van de ventilator wordt de bal omhoog geblazen. Met behulp van een microcontroller wordt de ventilator aangestuurd, zodat de bal op een gewenste hoogte blijft zweven. De regeling van de ventilatorsnelheid gebeurt momenteel met behulp van een PID-regelaar (Proportioneel-Integreer-Differentieel), waarbij de afwijking tussen de gewenste en werkelijke hoogte continu wordt bijgestuurd.

Hoewel deze opstelling functioneert zoals bedoeld, is het instellen van de PID-parameters vaak een handmatig proces. Het vergt tijd om het systeem goed af te stemmen om het uiteindelijk stabiel te laten reageren op hoogteveranderingen. Daarnaast kunnen veranderingen in de omgeving (zoals temperatuur of luchtvochtigheid) de prestaties van de PID-regelaar beïnvloeden.

De begeleidende docent heeft de vraag gesteld of deze traditionele regelmethode vervangen kan worden door een benadering gebaseerd op machine learning. Machine learning biedt de mogelijkheid om op basis van data te leren hoe het systeem zich moet gedragen. In dit project wordt onderzocht of het mogelijk is om een machine learning model te ontwikkelen dat het gedrag van de PID-regelaar kan nabootsen of zelfs verbeteren.

Dit onderzoek is relevant omdat het laat zien hoe klassieke regeltechniek vervangen kan worden door moderne, zelflerende systeem. Daarnaast sluit het aan bij actuele ontwikkelingen binnen de techniek en biedt het studenten inzicht in zowel embedded systemen als kunstmatige intelligentie. Uiteindelijk is het doel om een demonstratie-opstelling te creëren die gebruikt kan worden tijdens bijvoorbeeld open dagen, om zo op een toegankelijke manier beide technieken te tonen.

Het doel van dit onderzoek is om te onderzoeken of een neurale netwerk het gedrag van een PID-regelaar kan nabootsen of verbeteren in een systeem waarbij een pingpongbal op een ingestelde hoogte wordt gehouden. Hierbij wordt een model ontwikkeld, getraind en geëvalueerd in een gesimuleerde omgeving. Uiteindelijk moet het model geschikt zijn voor implementatie op een embedded platform, met als doel een stabiele en nauwkeurige hoogtecontrole te realiseren. Het onderzoek richt zich op zowel de technische haalbaarheid

als de vergelijking met de bestaande PID-regelaar. We gaan in dit onderzoek niet in op de implementatie van een neurale netwerk in een embedded omgeving, maar er wordt wel rekening gehouden met dat het uiteindelijk in een embedded omgeving geïmplementeerd moet worden. In dit onderzoek wordt de nadruk gelegd op of het mogelijk is om een neurale netwerk het PID-regelaar gedrag te laten nabootsen en hoe dit zich verhoudt tot de traditionele PID-regelaar.

Voor dit onderzoek wordt er rekening gehouden met het neurale netwerk op de **NXP FRDM-MCXN947 Development board** microcontroller. In het verloop van het onderzoek wordt er rekening gehouden met de beperkingen van deze microcontroller, zoals geheugen- en rekenkrachtbeperkingen. Dit is belangrijk omdat het uiteindelijke doel is om het getrainde model op deze microcontroller te implementeren. De NXP FRDM-MCXN947 Development board is gekozen omdat deze geschikt is voor embedded toepassingen en voldoende mogelijkheden biedt voor het uitvoeren van machine learning taken.

Voor dit onderzoek zijn er een aantal onderzoeks vragen geformuleerd die uiteindelijk de hoofdvraag van dit onderzoek moeten beantwoorden:

1. Hoe werkt een PID-regelaar en hoe wordt deze toegepast in de huidige opstelling?
2. Welke soorten neurale netwerken zijn geschikt voor regeltoepassingen in embedded systemen?
3. Hoe kan trainingsdata verzameld worden om het gedrag van de PID-regelaar na te bootsen?
4. Hoe presteert het getrainde neurale netwerk vergeleken met de traditionele PID-regelaar?
5. Wat zijn de beperkingen van het gebruik van een neurale netwerk in een embedded omgeving?

In dit rapport wordt eerst de benodigde theorie besproken (hoofdstuk 2), gevolgd door de gebruikte methoden (hoofdstuk 3) waarin we dit onderzoek doen. Daarna worden de resultaten gepresenteerd (hoofdstuk 4) en besproken (hoofdstuk 5). Het rapport sluit af met conclusies en aanbeveling (Hoofdstuk 6).

Aan het eind van het onderzoek zullen deze deelvragen worden beantwoord en onderbouwd, zodat de hoofdvraag kan worden beantwoord: *Is het mogelijk om een PID-regelaar te vervangen door een AI?*

2 Theorie

2.1 Werking van een PID-regelaar

Een PID-regelaar (Proportioneel-Integrerend-Differentiërend)^{[8][1]} is een veelgebruikte terugkoppelingsregelaar in de regeltechniek, die als doel heeft een systeem naar een gewenste waarde (setpoint) te sturen door continu het verschil (de fout) tussen die gewenste waarde en de gemeten systeemoutput te corrigeren. De PID-regelaar bestaat uit drie componenten:

- **Proportioneel (P):** Deze component reageert op de huidige fout. Hoe groter de fout, hoe sterker de aansturing. Dit zorgt voor een directe correctie, maar kan leiden tot een blijvende fout (steady-state error) als deze component alleen wordt gebruikt.
- **Integratorend (I):** Deze component reageert op de opgetelde (geïntegreerde) fout in de tijd. Hierdoor kan de regelaar kleine resterende fouten wegwerken en wordt het systeem naar het exacte setpoint geduwd. Te veel integratie kan echter leiden tot traagheid of instabiliteit.
- **Differentiërend (D):** Deze component reageert op de snelheid waarmee de fout verandert. Dit helpt om snelle veranderingen te dempen en voorkomt overshoot door het systeem te vertragen voordat het het setpoint bereikt. De D-term maakt het systeem reactiever en stabiel bij plotselinge veranderingen.

De regeloutput $u(t)$ van een PID-regelaar wordt meestal als volgt beschreven:

$$u(t) = K_p \cdot e(t) + K_i \cdot \int_0^t e(\tau) d\tau + K_d \cdot \frac{de(t)}{dt} \quad (1)$$

waarbij:

$e(t) = r(t) - y(t)$ (de fout tussen de referentie en gemeten waarde)

K_p = proportionele versterkingsfactor

K_i = integrerende versterkingsfactor

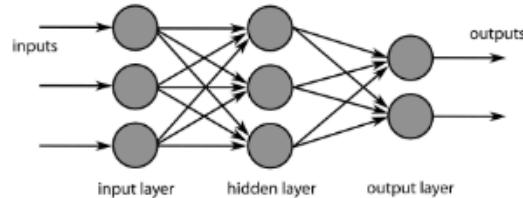
K_d = differentiërende versterkingsfactor

Een goed afgestemde PID-regelaar zorgt voor een snelle respons, minimale overshoot en een stabiele benadering van het setpoint zonder blijvende fout. Hoewel we in dit onderzoek niet in detail ingaan op de afstemming van PID-parameters, is het belangrijk te vermelden dat dit een cruciaal onderdeel is van het gebruik van PID-regelaars in de praktijk. De afstemming wordt meestal handmatig gedaan en dit is meestal een iteratief proces waarbij de waarden van K_p , K_i en K_d worden aangepast op basis van de prestaties van het systeem. Er zijn verschillende methoden voor PID-afstemming, zoals de Ziegler-Nichols-methode, maar deze vallen buiten de scope van dit onderzoek. In dit onderzoek richten we ons op het vervangen van de PID-regelaar door een neurale netwerk, waarbij we de nadruk leggen op het gedrag van de PID regelaar en de mogelijkheid om dit gedrag te repliceren met een neurale netwerk.

2.2 Neurale netwerken - basisprincipes

^{[2][7]}Een neurale netwerk is een wiskundig model dat is geïnspireerd op de werking van het menselijk brein. Het bestaat uit een verzameling onderling verbonden knopen, ook wel neuronen genoemd, die georganiseerd zijn in lagen: één inputlaag, één of meerdere verborgen lagen en één outputlaag. Elke verbinding tussen neuronen heeft een gewicht dat bepaalt hoe sterk een input wordt doorgegeven. Tijdens het doorlopen van het netwerk wordt elke input vermenigvuldigd met het bijbehorende gewicht, opgeteld met een bias en vervolgens door een activatiefunctie gehaald. Deze activatiefunctie bepaalt of en in

welke mate het neuron wordt geactiveerd. Het leerproces van een neurale netwerk gebeurt aan de hand van trainingsdata. Op basis van de fout tussen de voorspelde output en de werkelijke waarde worden de gewichten aangepast met behulp van een algoritme zoals bijvoorbeeld backpropagation in combinatie met gradient descent. Door herhaaldelijk trainen op veel voorbeelden leert het netwerk patronen en relaties herkennen in de data. Neurale netwerken zijn bijzonder krachtig in het modelleren van complexe, niet-lineaire systemen, en worden daarom veel gebruikt in toepassingen zoals beeldherkenning, spraakherkenning en regeltechniek.



Figuur 1: Illustratie van een neurale netwerk

2.3 vergelijking PID versus NN-regelaars

Zowel PID-regelaars als neurale netwerken kunnen worden ingezet om systemen te regelen, maar hun werking, toepassingsgebied en eigenschappen verschillen sterk.

2.3.1 Verschillen

- **Regelprincipe:** Een PID-regelaar is gebaseerd op een vaste wiskundige formule die werkt met de fout tussen gewenste en gemeten waarden. Een neurale netwerk leert juist op basis van voorbeelddata een model van het systeemgedrag.
- **Aanpasbaarheid:** PID-regelaars hebben vaste parameters K_p , K_i , K_d die handmatig of via tuning bepaald worden. Neurale netwerken leren automatisch complexe relaties tijdens het trainen, zonder expliciet geprogrammeerde regels.
- **Complexiteit:** PID-regelaars zijn relatief eenvoudig te implementeren en begrijpen. Neurale netwerken zijn complexer en vereisen meer rekenkracht en data.
- **Flexibiliteit:** PID is minder geschikt voor sterk niet-lineaire systemen of systemen met veel vertraging. Neurale netwerken kunnen deze situaties beter aan, mits voldoende training.
- **Transparantie:** PID-regelaars zijn goed uitlegbaar. Neurale netwerken zijn vaak 'black boxes', waarbij het moeilijk is om te achterhalen waarom een bepaalde beslissing wordt genomen.

2.3.2 Overeenkomsten

- Beide regelmethoden kunnen gebruikt worden voor het aansturen van dynamische systemen.
- Beide maken gebruik van foutsignalen (in neurale netwerken impliciet tijdens het trainen).
- Beide kunnen met feedback werken om de systeemoutput te corrigeren.
- Beide vereisen afstemming of training om goed te presteren in een specifieke toepassing.

2.4 keuze van netwerkarchitecturen

Voordat er in gegaan wordt op de NN-architecturen voor deze toepassing is het belangrijk om met een aantal factoren rekening mee te houden. Deze factoren zijn van belang bij het kiezen van een geschikte NN-architectuur voor het vervangen van een PID-regelaar in een embedded systeem. De factoren die onder andere worden besproken zijn:

- **Real-time vereisten:** Als het systeem real-time reacties vereist, moet de gekozen architectuur snel genoeg zijn om aan deze eisen te voldoen. Dit kan de keuze beperken tot lichtere netwerken of optimalisaties zoals quantisatie.
- **Ondersteuning voor embedded systemen:** De gekozen architectuur moet compatibel zijn met de hardware van het embedded systeem, zoals microcontrollers of FPGA's. Dit kan beperkingen opleggen aan de complexiteit en het type netwerk.
- **Ondersteuning voor machine learning frameworks:** Het is belangrijk om te kiezen voor machine learning frameworks die ondersteuning bieden voor embedded systemen, zoals TensorFlow Lite, Keras of PyTorch Mobile. Deze frameworks zijn ontworpen om neurale netwerken te optimaliseren en te implementeren op beperkte hardware.
- **Trainings- en implementatietijd:** De tijd die nodig is om het neurale netwerk te trainen en te implementeren, kan ook een factor zijn bij de keuze van de architectuur. Sommige netwerken vereisen meer tijd voor training en afstemming dan andere.

Aan de hand van deze factoren kunnen we een weloverwogen keuze maken voor de netwerkarchitectuur die het beste past bij de specifieke toepassing en de vereisten van het systeem. Het is belangrijk om de juiste balans te vinden tussen complexiteit, prestaties en implementatiegemak, zodat het neurale netwerk effectief kan functioneren als vervanging voor de PID-regelaar. Hieronder worden enkele architecturen die overwogen kunnen worden voor het vervangen van een PID-regelaar door een neurale netwerk met hun voor- en nadelen:

- **Recurrente netwerken^[3] (RNN):** Deze netwerken zijn ontworpen om tijdsafhankelijke data te verwerken door informatie van eerdere tijdstappen te onthouden. Ze zijn nuttig voor systemen waarbij de huidige output afhankelijk is van eerdere inputs, zoals bij dynamische systemen. RNN's kunnen echter complexer zijn om te trainen en vereisen meer rekenkracht.
- **Convolutioneel Neuraal Netwerk^[4] (1D CNN):** Een 1D CNN is specifiek geschikt voor sequentiële of tijdsgebaseerde data, zoals signalen (bijvoorbeeld audio of ECG), tijdreeksen (zoals beursdata of temperatuurmetingen) en reeksen met meerdere features (zoals sensorwaarden). Hoewel convolutionele netwerken vooral bekend zijn uit de beeldverwerking, kunnen 1D CNN's effectief patronen en lokale afhankelijkheden in tijdreeksen herkennen, waardoor ze ook toepasbaar zijn voor regeltoepassingen waarbij de input bestaat uit opeenvolgende meetwaarden.
- **Long Short-Term Memory (LSTM):** Dit is een type RNN dat speciaal is ontworpen om lange-termijn afhankelijkheden te onthouden. LSTM's zijn zeer geschikt voor systemen met complexe dynamiek en tijdsafhankelijke gedragspatronen, maar ze zijn ook complexer en vereisen meer data om goed te presteren.

Voor de toepassing van het vervangen van een PID-regelaar zijn er aantal kenmerken die belangrijk zijn bij het kiezen van een geschikte netwerkarchitectuur:

- **Eenvoudige structuur:** Voor een microcontroller is het belangrijk dat de netwerkarchitectuur eenvoudig genoeg is om

efficiënt te trainen en te implementeren. Complexe netwerken zoals LSTM's kunnen te veel rekenkracht vereisen en zijn moeilijker te optimaliseren voor embedded systemen.

- **Compact model:** Het model moet klein genoeg zijn om op de microcontroller te passen, wat betekent dat het aantal parameters beperkt moet zijn. Dit kan worden bereikt door het aantal lagen en neuronen te beperken, of door technieken zoals quantisatie toe te passen.
- **Snelle inferentie:** De gekozen architectuur moet snel genoeg zijn om real-time voorspellingen te doen, wat betekent dat de inferentietijd laag moet zijn. Dit kan worden bereikt door gebruik te maken van efficiënte activatiefuncties en optimalisaties in de netwerkstructuur.
- **Robuustheid:** Het netwerk moet robuust zijn tegen verstoringen en veranderingen in de systeemparameters. Dit kan worden bereikt door het netwerk te trainen op een breed scala aan scenario's en variaties in de data.
- **Ondersteuning voor embedded systemen:** De gekozen architectuur moet compatibel zijn met de hardware van het embedded systeem, zoals microcontrollers of FPGA's. Dit kan beperkingen opleggen aan de complexiteit en het type netwerk.
- **Ondersteuning voor machine learning frameworks:** Het is belangrijk om te kiezen voor machine learning frameworks die ondersteuning bieden voor embedded systemen, zoals TensorFlow Lite, Keras of PyTorch Mobile. Deze frameworks zijn ontworpen om neurale netwerken te optimaliseren en te implementeren op beperkte hardware.
- **Flexibiliteit in tijdsvenster:** De architectuur moet in staat zijn om te werken met verschillende tijdsvensters van inputdata, zodat het netwerk kan worden aangepast aan de specifieke toepassing en de vereisten van het systeem.

Deze kenmerken helpen bij het selecteren van een netwerkarchitectuur die geschikt is voor het vervangen van een PID-regelaar in een embedded systeem. Het is belangrijk om de juiste balans te vinden tussen complexiteit, prestaties en implementatiegemak, zodat het neurale netwerk effectief kan functioneren als vervanging voor de PID-regelaar.

uiteindelijk is er gekozen voor een Convolutioneel Neuraal Netwerk (1D CNN) als de meest geschikte architectuur voor het vervangen van een PID-regelaar in een embedded systeem. Deze keuze is gebaseerd op de volgende overwegingen:

- **Eenvoudige structuur:** Een 1D CNN heeft een overzichtelijke en relatief eenvoudige architectuur, waardoor het netwerk efficiënt te trainen en te implementeren is, zeker in vergelijking met complexere netwerken zoals LSTM's.
- **Compact model:** 1D CNN's kunnen worden geoptimaliseerd tot compacte modellen met een beperkt aantal parameters, wat ze geschikt maakt voor embedded systemen met beperkte rekenkracht en geheugen.
- **Snelle inferentie:** Door hun efficiënte structuur bieden 1D CNN's doorgaans snellere inferentie dan recurrente netwerken, wat essentieel is voor real-time toepassingen.
- **Robuustheid en generalisatie:** 1D CNN's zijn effectief in het herkennen van patronen en lokale afhankelijkheden in sequentiële data, waardoor ze goed kunnen generaliseren naar verschillende scenario's en bestand zijn tegen verstoringen.
- **Geschikt voor embedded systemen:** Er zijn diverse machine learning frameworks, zoals TensorFlow Lite en Keras, die 1D

CNN's ondersteunen en optimaliseren voor gebruik op embedded hardware.

- **Brede frameworkondersteuning:** 1D CNN's worden breed ondersteund door populaire machine learning frameworks, wat de ontwikkeling, optimalisatie en implementatie vereenvoudigt.
- **Flexibiliteit in tijdsvenster:** Met 1D CNN's kan eenvoudig worden bepaald over welk tijdsvenster inputdata wordt geanalyseerd, waardoor het netwerk flexibel inzetbaar is voor uiteenlopende regeltoepassingen.

Deze overwegingen maken de 1D CNN een geschikte keuze voor het vervangen van een PID-regelaar in een embedded systeem. Het biedt een goede balans tussen prestaties, eenvoud en implementatiegemak, waardoor het effectief kan functioneren als vervanging voor de PID-regelaar in verschillende toepassingen.

2.5 Vergelijking van prestaties

De prestaties van een neuraal netwerk in vergelijking met een PID-regelaar kunnen op verschillende manieren worden beoordeeld, afhankelijk van de specifieke toepassing en de doelstellingen van de regeling. Enkele belangrijke prestatie-indicatoren zijn:

- **Nauwkeurigheid:** Hoe goed het neuraal netwerk de gewenste output kan voorspellen in vergelijking met de PID-regelaar. Dit kan worden gemeten aan de hand van de fout tussen de voorspelde en werkelijke waarden.
- **Stabiliteit:** Hoe stabiel het systeem is bij het bereiken van de gewenste output. Een PID-regelaar kan soms oscillaties vertonen, terwijl een goed getraind neuraal netwerk deze kan verminderen of elimineren.
- **Snelheid:** Hoe snel het neuraal netwerk kan reageren op veranderingen in de input. Dit is vooral belangrijk in real-time toepassingen waar snelle aanpassingen nodig zijn.
- **Robuustheid:** Hoe goed het neuraal netwerk presteert onder verschillende omstandigheden, zoals verstoringen of veranderingen in de systeemparameters. Een PID-regelaar kan soms gevoelig zijn voor veranderingen in de omgeving, terwijl een neuraal netwerk mogelijk beter kan generaliseren.

De vergelijking van prestaties tussen een neuraal netwerk en een PID-regelaar kan worden uitgevoerd door beide systemen te testen op dezelfde set van inputs en de resultaten te vergelijken. Dit kan bijvoorbeeld worden gedaan door simulaties uit te voeren of door het systeem in de praktijk te testen. Het is belangrijk om de prestaties onder verschillende omstandigheden te evalueren, zoals bij verstoringen of veranderingen in de systeemparameters, om een goed beeld te krijgen van de effectiviteit van het neuraal netwerk in vergelijking met de PID-regelaar.

2.6 Implementatie op embedded systemen

Een van de uitdagingen van het implementeren van een neuraal netwerk is de integratie ervan in embedded systemen. Embedded systemen zijn vaak beperkt in termen van rekenkracht, geheugen en energieverbruik, wat betekent dat de implementatie van een neuraal netwerk op deze systemen specifieke overwegingen vereist. Daarom is het belangrijk om de volgende aspecten in overweging te nemen bij de implementatie van een neuraal netwerk op embedded systemen:

- **Modeloptimalisatie:** Het neuraal netwerk moet worden geoptimaliseerd voor de beperkte rekenkracht en geheugen van het embedded systeem. Dit kan onder meer het verminderen van het aantal parameters, het gebruik van quantisatie of het toepassen van modelcompressie omvatten.

• **Efficiënte inferentie:** De inferentie (voorspellingsfase) van het neuraal netwerk moet efficiënt worden uitgevoerd. Dit kan worden bereikt door gebruik te maken van speciale hardwareversnellers, zoals Tensor Processing Units (TPU's) of Field Programmable Gate Arrays (FPGA's), die zijn ontworpen voor het uitvoeren van neurale netwerken.

• **Real-time prestaties:** Het neuraal netwerk moet in staat zijn om real-time voorspellingen te doen, wat betekent dat de inferentietijd snel genoeg moet zijn om te voldoen aan de vereisten van de toepassing. Dit kan worden bereikt door het netwerk te optimaliseren voor snelheid en door gebruik te maken van efficiënte algoritmen.

• **Compatibiliteit met hardware:** Het neuraal netwerk moet compatibel zijn met de specifieke hardware van het embedded systeem. Dit kan onder meer het gebruik van specifieke bibliotheken of frameworks omvatten die zijn geoptimaliseerd voor de gekozen hardware.

• **Ondersteuning voor machine learning frameworks:** Het is belangrijk om te kiezen voor machine learning frameworks die ondersteuning bieden voor embedded systemen, zoals TensorFlow Lite, Keras of PyTorch Mobile. Deze frameworks zijn ontworpen om neurale netwerken te optimaliseren en te implementeren op beperkte hardware.

Deze microcontroller is geschikt voor embedded toepassingen en biedt voldoende mogelijkheden voor het uitvoeren van machine learning taken. Hieronder volgt een uitgebreid overzicht van de belangrijkste kenmerken van de NXP FRDM-MCXN947^[5] Development board, gebaseerd op de officiële NXP-pagina:

• **Processor:** De FRDM-MCXN947^[5] is uitgerust met een dual-core ARM Cortex-M33 processor, draaiend op maximaal 150 MHz per core. Deze cores zijn ontworpen voor deterministische prestaties en ondersteunen TrustZone-technologie voor beveiligde toepassingen. De gecombineerde rekenkracht (750 DMIPS per core) maakt deze geschikt voor besturingsslogica, preprocessing en embedded AI-taken.

• **Neural Processing Unit (NPU):** De geïntegreerde eIQ Neutron NPU biedt tot 16 GOPS (Giga Operations Per Second) rekenkracht voor machine learning-inferentie. De NPU versnelt modellen met tientallen keren vergeleken met CPU-only uitvoering. Bijvoorbeeld: een keyword spotting-model van 30 kB haalt inferentietijden van enkele milliseconden met een nauwkeurigheid van 97.06%.

• **Geheugen:** De MCU bevat 2 MB on-chip flash in dual-bank configuratie, geschikt voor firmware updates tijdens runtime. Daarnaast is er 512 kB SRAM beschikbaar. Voor grotere modellen of datasets kan via SPI een externe micro-SD-kaart worden toegevoegd.

• **Machine learning frameworks:** TensorFlow Lite for Microcontrollers en het NXP eIQ framework worden ondersteund. Deze frameworks maken optimalisatie via quantisatie, pruning en compressie mogelijk, essentieel voor inferentie op embedded hardware.

• **Real-time prestaties:** Dankzij een systeemtimer, nested vectored interrupt controller (NVIC), DMA-ondersteuning en hardwarematige versnelling van AI-taken kan het systeem real-time inferenties uitvoeren (bijvoorbeeld 1–5 ms per inferentie bij kleine modellen).

Deze kenmerken maken de NXP FRDM-MCXN947 Development board een geschikte keuze voor het implementeren van een neuraal netwerk als vervanging voor een PID-regelaar in embedded systemen. Door rekening te houden met de mogelijkheden en beperkingen

van deze microcontroller, kan het neurale netwerk worden geoptimaliseerd voor de specifieke toepassing en de vereisten van het systeem.

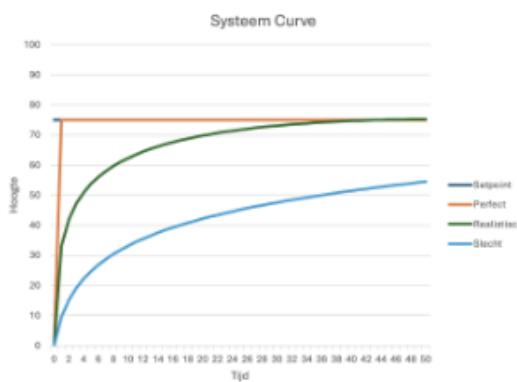
3 Methode

In dit hoofdstuk wordt beschreven op welke wijze het onderzoek is uitgevoerd. De nadruk ligt op de gebruikte software, de strategie voor dataverzameling, de opbouw en training van het neurale netwerk, en de testomgeving waarin de prestaties van het model zijn geëvalueerd. Er is specifiek rekening gehouden met de beperkingen van de NXP FRDM-MCXN947 microcontroller.

3.1 Manieren van dataverzameling

Het verzamelen van trainingsdata vormt een cruciale stap in het opzetten van een neurale netwerk voor regeltoepassingen. De gekozen methode heeft directe invloed op de prestaties van het model. Binnen dit onderzoek zijn drie methodes onderzocht voor het trainen van de AI:

1. **PID-kopie:** Het neurale netwerk wordt getraind op data afkomstig van een werkende PID-regelaar. Het doel is om het gedrag van de PID exact na te bootsen. Deze aanpak vereist een goed afgestemde PID als referentie en resulteert in een AI-model dat functioneert binnen dezelfde prestatiegrenzen als de oorspronkelijke regelaar. Een nadeel is de benodigde werkende PID.
2. **Ideale responscurve:** Hierbij wordt de AI getraind op een theoretisch bepaalde ideale curve, waarin de hoogte van de pingpongbal over de tijd een gewenst verloop volgt (zie bijvoorbeeld zie Figuur 2). Dit biedt meer flexibiliteit bij het verbeteren van de prestaties, maar vereist een nauwkeurige definitie van het gewenste gedrag. Een nadeel is dat kleine afwijkingen in setpoints relatief trage correcties veroorzaken in vergelijking met grote setpointveranderingen.
3. **Iteratieve benadering:** Deze methode combineert de bovenstaande methoden. In de eerste fase wordt het netwerk getraind op data van een PID. Vervolgens worden verbeteringen aangebracht met behulp van de ideale curve. Deze methode maakt snelle en gecontroleerde iteratieve verbeteringen mogelijk.



Figuur 2: manieren van dataverzameling

De initiële dataset werd gegenereerd door middel van simulaties, waarin een bestaande PID-regelaar werd nagebootst. Dit leverde gestructureerde gegevens op, waaronder: tijdstap, setpoint, werkelijke hoogte, fout (setpoint - gemeten waarde), geïntegreerde fout, afgelide fout en de bijbehorende PWM-waarde. Het uiteindelijke

doel is om het netwerk ook te trainen met data uit een fysieke opstelling. Simulatiedata zijn reproduceerbaar en snel te genereren, maar missen systeemeigenschappen die voortkomen uit hardware-imperfecties. Door aanvullend gebruik te maken van data uit de echte opstelling kunnen robuustere modellen worden ontwikkeld. Een bruikbare dataset moet representatief zijn voor diverse systeemcondities. Bij dataverzameling uit de reële opstelling is daarom gelet op variatie in setpoints, verstoringen en afwijkingen in hardwarecomponenten. Het gebruik van meerdere hardware opstellingen elimineert ook de verschillende afwijkingen in hardware en zou dus toepasbaar moeten zijn op verschillende opstellingen. Parameters die consequent zijn meegenomen in de dataset zijn: het setpoint, de actuele hoogte van de bal, de fout, de geïntegreerde fout en de afgelide fout.

3.2 Software en tools

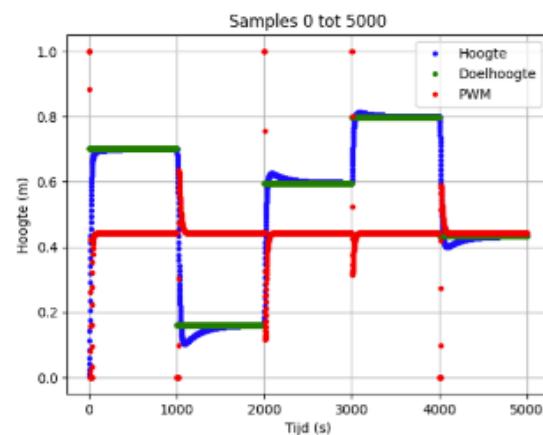
Voor zowel de simulatie van de pingpongbal-opstelling als het opzetten en trainen van het neurale netwerk is gebruik gemaakt van Python. De volgende libraries en tools zijn ingezet:

- **NumPy** en **Pandas**: voor het verwerken en analyseren van data;
- **Matplotlib** voor visualisatie van resultaten;
- **TensorFlow**^[6] en **Keras**: voor het bouwen, trainen en valideren van het neurale netwerk;
- **Scikit-learn**: voor data preprocessing en evaluatiemethoden;
- **Jupyter Notebook**: voor het ontwikkelen en testen van de code in iteraties.

3.3 Dataverzameling

De trainingsdata voor het neurale netwerk is verzameld door een simulatie van de bestaande PID-regelaar. In deze simulatie werd het gedrag van de PID-regelaar gelogd terwijl deze een pingpongbal op verschillende hoogtes stabiel probeerde te houden. Hierbij zijn gegevens zoals tijd, doelhoogte, werkelijke hoogte, fout, fout integratie, fout afgelide en bijbehorende PWM-waarden opgeslagen (zie Figuur 3).

De data werd vervolgens gesplitst in een trainingsset (80%) en een validatieset (20%). De trainingsset werd gebruikt om het model te trainen, terwijl de validatieset diende om de prestaties van het model te evalueren tijdens de training. Dit helpt overfitting te voorkomen en zorgt ervoor dat het model generaliseerbaar blijft naar nieuwe data.



Figuur 3: PID gesimuleerde data

3.4 Neuraal netwerk opzetting

Op basis van de kenmerken van het systeem en de vereisten voor embedded implementatie is gekozen voor een 1D Convolutioneel Neuraal Netwerk (1D CNN). Dit type netwerk is efficiënt in het herkennen van patronen in sequentiële data en heeft relatief lage rekenverrichtingen.

De architectuur van het netwerk bestaat uit:

- Een inputlaag met vensters van opeenvolgende foutwaarden;
- Een of meerdere 1D convolutionele lagen met ReLU-activatie;
- Een flatten-laag gevolgd door één dichte laag;
- Een outputlaag die de benodigde PWM-waarde voorspelt.

De volledige code van het neuraal netwerk is openbaar beschikbaar op GitHub.* De code is geschreven in Python en maakt gebruik van TensorFlow en Keras voor de implementatie van het neuraal netwerk. Daarnaast is er een handleiding toegevoegd, zodat gebruikers eenvoudig stap voor stap door het proces kunnen lopen en het netwerk zelf kunnen opzetten en trainen.

3.5 Trainingsstrategie en hyperparameters tuning

Het model is getraind met behulp van backpropagation en de Adam optimizer. De volgende hyperparameters zijn getest en afgestemd:

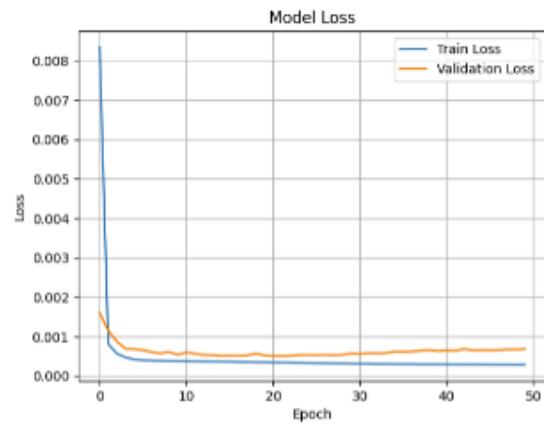
- Learning rate: 0.001, 0.0005, 0.0001;
- Batch size: 32 en 64;
- Aantal epochs: 10–100;
- Rechthoekige activatievensters: lengte 5–20 samples.

De performance van het model werd tijdens de training geëvalueerd met behulp van de mean squared error (MSE). Early stopping werd toegepast om overfitting te voorkomen.

3.6 Testopstelling of Simulatieomgeving

De opstelling is volledig gesimuleerd in Python, waarbij de dynamiek van de pingpongbal in een verticale buis werd gemodelleerd. Hierbij werd rekening gehouden met de krachten op de bal (zwaartekracht, luchtdruk door de ventilator) en de vertraging in het systeem. Zowel de PID-regelaar als het neuraal netwerk werd getest in dezelfde simulatieomgeving zodat een eerlijke vergelijking kon worden gemaakt.

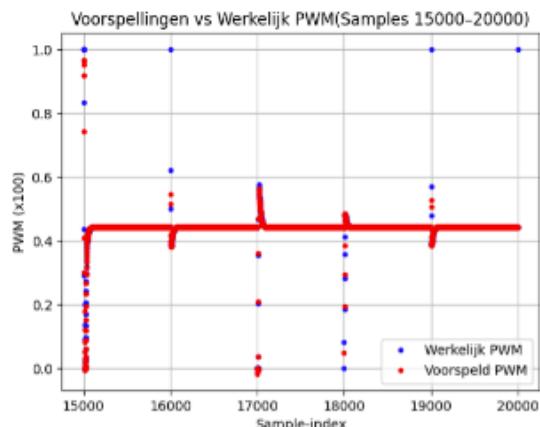
Het gedrag van beide regelstrategieën werd geëvalueerd op responsietijd, stabiliteit, overshoot, en robuustheid bij verstoringen.



Figuur 4: Training en validation loss van het neuraal netwerk

4.2 vergelijking met de PID-regelaar

Om het model te kunnen vergelijken met de traditionele PID-regelaar, zijn beide regelstrategieën getest in dezelfde simulatieomgeving. Hierbij werd gekeken naar de prestaties op responsietijd, stabiliteit, overshoot en robuustheid. In figuur 5 is een overzicht opgenomen van de meetresultaten. Hieruit blijkt dat het getrainde neuraal netwerk een vergelijkbare prestatie levert als de PID-regelaar in termen van hoogtecontrole van de pingpongbal. Het model wist de gewenste hoogte nauwkeurig te benaderen met een vergelijkbare stabiliteit en minder overshoot in sommige scenario's. De resultaten van deze vergelijking zijn tevens weergegeven



Figuur 5: Vergelijking van de prestaties van het neuraal netwerk en de PID-regelaar

4.3 Effect van variatie in data/robustheid

In de eerste iteratie van het model werd gekozen voor een grotere netwerkarchitectuur met meerdere verborgen lagen en neuronen per laag. Na overleg met een expert uit het Datalab is besloten om deze complexiteit te reduceren. De reden hiervoor was niet gelinkt aan de beperkingen van de microcontroller, maar aan de aard van het regelprobleem zelf. Het bleek dat het gedrag van een PID-regelaar relatief eenvoudig te modelleren is, waardoor een kleiner model efficiënter betere prestaties leverde. Bovendien werd vastgesteld dat een kleinere architectuur minder gevoelig is voor overfitting en beter presteert onder variërende invoerdata. Dit verhoogt de robuustheid van

*https://github.com/MITCHEL-Development/PEE51---AIRegelsysteem/tree/main/03_NN-Handleiding

het model, wat essentieel is bij implementatie op fysieke hardware.

4.4 Observatie tijdens simulatie

Tijdens de simulaties viel op dat het neurale netwerk in staat was om consistente prestaties te leveren. De stabiliteit en reactietijd bleven binnen aanvaardbare marges. Hoewel deze observaties grotendeels overeenkomen met de bevindingen uit de vorige paragrafen, bevestigen ze de geschiktheid van het model voor real-time regeltoepassingen.

- De prestaties van het model kwamen sterk overeen met die van de PID-regelaar.
- Het model is compact genoeg voor implementatie op een embedded systeem.
- De keuze voor een eenvoudige netwerkarchitectuur leidt zowel tot snelheid als robuustheid.
- Validatie met gesimuleerde data toont aan dat de AI in staat is tot stabiele en nauwkeurige regeling.

5 Discussie

5.1 Interpretatie van de resultaten

De verkregen resultaten tonen aan dat een neurale netwerk in staat is om het gedrag van een PID-regelaar met hoge nauwkeurigheid na te bootsen. Daarbij is ook rekening gehouden met de hardware beperkingen van de microcontroller. Bovendien maakt het ontwikkelde VSC-bestand het mogelijk om eenvoudig variaties van het netwerk te implementeren of te testen met andere datasets. Dit biedt veel flexibiliteit voor toekomstige optimalisaties of uitbreidingen van het systeem.

5.2 Beperkingen van het model

De voornaamste beperkingen liggen bij de beschikbare hardware. Zoals besproken in hoofdstuk 2.6 is het geheugen op de microcontroller beperkt, wat het gebruik van grotere of meer geavanceerde modellen, zoals LSTM-netwerken, belemmert. Ook is er geen rekening gehouden met langdurige inzet of prestaties onder extreme omstandigheden, omdat het model nog niet fysiek is getest.

5.3 Lessen en inzichten

Gedurende het project werd duidelijk dat eenvoudige modelontwerp vaak gunstiger is, zeker in embedded toepassingen. Daarnaast is het belang van kwaliteit, divers en goed gestructureerde trainingsdata niet te onderschatten. Een iteratieve aanpak in modelontwikkeling en code generatie blijkt essentieel om gericht te verbeteren.

5.4 Reflectie op de aanpak

De gekozen methode – waarbij gestart werd met een kopie van de PID en vervolgens verfijning plaatsvond via trainingsdata – bleek tot huidige prestaties effectief. Het combineren van simulatie en validatie leverde waardevolle inzichten op, terwijl de samenwerking met experts uit het Databab zorgde voor praktische richtlijnen. Wel hadden sommige onderdelen van de code of trainingsstrategie mogelijk eerder geoptimaliseerd kunnen worden.

6 Conclusie en Aanbevelingen

6.1 Beantwoording van de hoofdvraag

Is het mogelijk om een PID-regelaar te vervangen door een AI?

Uit de resultaten van dit onderzoek blijkt dat een neurale netwerk het gedrag van een PID-regelaar succesvol kan nabootsen. De AI-regelaar presteert vergelijkbaar met de traditionele PID in een gesimuleerde omgeving.

6.2 Samenvatting van de belangrijkste bevindingen

- Een **1D Convolutioneel Neuraal Netwerk (1D CNN)** bleek het meest geschikt voor deze toepassing.

6.3 Aanbevelingen voor implementatie

Hoewel TensorFlow Lite ondersteuning biedt voor embedded toepassingen, bleek de EIQ-software van NXP primair gericht op beeldherkenning. Voor praktische implementatie is het daarom aanbevolen om de meegeleverde voorbeeldcode van de NXP FRDM-MCXN947 als basis te gebruiken en deze iteratief, met bijvoorbeeld trunk based development, uit te breiden met het getrainde model. Zo blijft het systeem controleer- en schaalbaar.

6.4 Suggesties voor vervolgonderzoek

Voor vervolgonderzoek wordt het volgende aanbevolen:

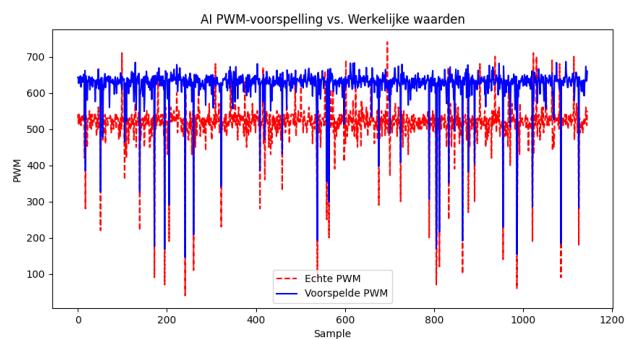
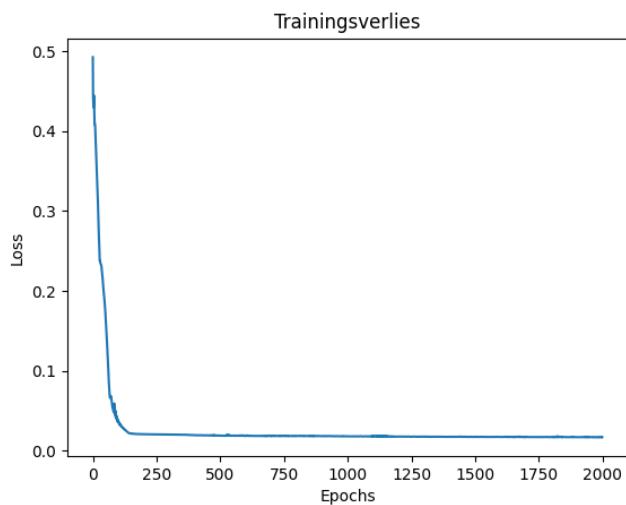
- Het model fysiek implementeren op de microcontroller en de real-time prestaties evalueren.
- Verschillende vormen van trainingsdata vergelijken (reëel vs. simulatie).
- De impact van verstoringen, ruis en sensorgebreken op de robuustheid van het model analyseren.
- Experimenteren met andere netwerkarchitecturen, zoals LSTM's, mits de hardware dit toelaat.

Referenties

- [1] Stan Franklin. *Artificial Minds*. MIT Press, Cambridge, MA, 1995. Zie p. 293.
- [2] GeeksforGeeks. Recurrent neural networks (rnn) - geeksforgeeks, 2024. Accessed: 2024-06-01.
- [3] Simon Haykin. *Neural Networks and Learning Machines*. Pearson Education, Upper Saddle River, NJ, 3rd edition, 2009. Zie p. 23.
- [4] Simon Haykin. *Neural Networks and Learning Machines*. Pearson Education, Upper Saddle River, NJ, 3rd edition, 2009. Zie p. 201.
- [5] NXP Semiconductors. MCXNx4x Family: 32-bit Arm Cortex-M33 @ 150 MHz (Datasheet) (N94x and N54x). <https://www.nxp.com/docs/en/data-sheet/MCXNx4xDS.pdf>, 2025. Product Data Sheet, Rev. 7 — 25 January 2025.
- [6] TensorFlow. Tensorflow tutorials, 2024. Accessed: 2024-06-14.
- [7] Abhishek Vijayvargia. *Machine Learning with Python: Design and Develop Machine Learning and Deep Learning Systems Using Real-World Examples*. BPB Publications, 2018.
- [8] Wikipedia contributors. Pid-regelaar — wikipedia, de vrije encyclopedie, 2024. Accessed: 2024-06-01.

UITSLAGEN COMPLEXE AI'S

EERSTE TOEPASSING



Simple RNN, 2 verborgen lagen, met 2

