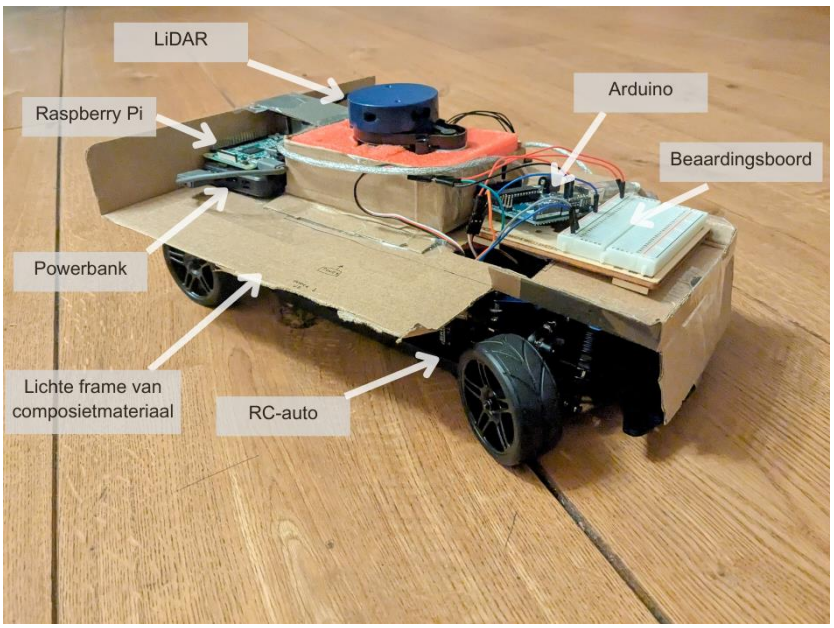


ZELFRIJDENDE AUTO

Met AI als bestuurder

Shengen Zhou, Lucas de Goeij

Willem Denijs

17 Januari 2025

Inhoudsopgave

H1 – Voorwoord 4
H2 – Inleiding 5
2.1 Deelvragen 5
2.2 Programma van eisen 6
H3 - Kunstmatige intelligentie 7
3.1 Neuraal netwerk 7
3.1A Hoe werkt een neuraal netwerk? 7
3.1B Wat gebeurt er in een neuraal netwerk? 7
3.1C Wat is de input/output van ons neuraal netwerk? 8
3.1D Hoe ziet ons neuraal netwerk eruit? 9
3.2 Machine learning 10
3.2A Waarom NEAT? 10
3.2B NEAT machine learning algoritme 10
3.2C Het concept van NEAT (Theoretisch kader) 10
3.2D Mutatie NEAT (Theoretisch kader) 10
3.2E Cross-overs NEAT (Theoretisch kader) 11
3.2F Speciation NEAT (theoretisch kader) 12
3.3 Toepassen machine learning 14
3.3A Hoe trainen we onze neuraal netwerken? 14
3.3B Wat is onze fitness functie? 14
H4 Simulatie 15
4.1 Structuur van de code 15
4.1A Klassen 15
4.1B “Car” klasse 15
4.1C De lidar klasse 22
4.1D het main bestand 24
4.2 Realistisch simuleren 28
H5 Ontwerp zelfrijdende auto 29

5.1 Hoe werkt onze zelfrijdende auto? 29
5.1A Wat is een API? (Theoretisch kader) 30
5.2 LiDAR sensor 31
5.2A Hoe werkt de LiDAR sensor? 31
5.3 Raspberry Pi 32
5.3A Hoe verwerkt de Raspberry Pi de LiDAR afstandspunten? 32
5.3C Versimpelde code op de Raspberry pi 32
5.4 GPIO 34
5.4A Apparaten verbinden met GPIO 34
5.4B Beaardingsboord 34
5.5 Arduino aansluiten aan servomotoren 35
5.5A Hoe sluiten we de Arduino aan op de servomotor? 35
5.5B Hoe werkt een servomotor? 35
5.6 Arduino 36
5.6A Hoe ontvangen we en sturen we signalen? 36
H6 Testen (Werkwijze) 37
6.1 Hoe gaan we testen? 37
6.1A Testen Arduino en servomotoren 37
6.1B Testen LiDAR en Raspberry Pi 37
6.1C Testen LiDAR, Raspberry Pi, Arduino, servomotoren, RC auto 38
6.1D Testen zelfrijdende auto 39
H7 Resultaten 40
7.1 Resultaten simulatie 41
7.1A Fitnesswaarden 41
7.1B Leert het beste model (Daniel v3) wel echt autorijden? 42
7.2 Resultaten zelfrijdende auto 43
H8 Discussie 45
8.1 Discussie simulatie 45
8.2 Discussie zelfrijdende auto 45

H9 – Conclusie 47
H10 – Reflectie 48
Logboek 49
Bronvermelding 50

H1 – Voorwoord

Onze motivatie om een zelfrijdende auto te maken kwam vooral omdat wij onszelf veel wouden uitdagen met iets heel moeilijks. Hierdoor hebben wij de afgelopen maanden heel veel geleerd over software, hardware en systemen integreren. Daarnaast zijn we heel dankbaar voor onze ouders die voor ons project nieuwe hardware hebben gekocht. Ten slotte willen we Willem Denijs nog bedanken voor het begeleiden.

Wij wensen je veel leesplezier!

H2 – Inleiding

Sinds het begin van tijd proberen mensen steeds op uitvindingen te komen die ze minder hoeven te laten doen. Eerst het wiel, om dingen op te vervoeren. Daarna kwam agricultuur, zodat ze niet meer hoefden te jagen. Vele jaren later kwam de industriële revolutie, machines deden nu de handmatige arbeid waar mensen eerst zo veel moeite in moesten stoppen. Sinds kort proberen mensen zelfrijdende auto's te bouwen, die ons van onze taak van het rijden gaat bevrijden. Wij wouden een bijdrage maken en komen daarom met de Daniel v3.

Om een zelfrijdende auto te ontwerpen hebben we veel verschillende soorten vaardigheden geleerd. Zo moeten we ten eerste een simulatieomgeving programmeren om een eigen AI-model te trainen. Verder gaan we met cutting edge technology werken zoals een LiDAR sensor, Raspberry Pi en Arduino, om deze hardware te kunnen gebruiken leren we veel technische vaardigheden. Ten slotte hebben we ook natuurkundige kennis nodig om de hardware aan te sluiten aan de RC-auto.

In dit verslag gaan we antwoord geven op onze hoofdvraag:

Hoe maken we een zelfrijdende auto?

2.1 Deelvragen

Om antwoord te geven op onze hoofdvraag hebben we de volgende deelvragen opgesteld, deze deelvragen gaan we beantwoorden in de volgende hoofdstukken.

- Hoe werkt een neuraal netwerk?
- Wat gebeurt er in een neuraal netwerk?
- Wat is de input/output van ons neuraal netwerk?
- Hoe ziet ons neuraal netwerk eruit?
- Hoe trainen we onze neuraal netwerken met NEAT?
- Wat gebruiken we als fitness functie?
- Hoe gaan we de code structureren voor de simulatie?
- Hoe simuleren we realistisch?
- Hoe werkt onze zelfrijdende auto?
- Hoe werkt een LiDAR sensor?
- Hoe verwerkt de Raspberry Pi de LiDAR afstandspunten?
- Hoe stuurt de Raspberry Pi de signalen naar de Arduiino?
- Hoe verbind je twee apparaten met GPIO?
- Hoe sluiten we de Arduino aan op de servomotoren?

- Hoe ontvangt en stuurt de Arduino signalen?
- Hoe gaan we testen?
- Leert het model wel echt autorijden?

2.2 Programma van eisen

Ons grootste doel was om de auto zelfstandig een baan zonder enige botsingen af te kunnen laten leggen. Om dit te bereiken hebben we gekozen om het model te trainen op gesimuleerde banen met de hulp van AI, in plaats van een vast algoritme. In een vast algoritme zou je het hard gecodeerd in je programma moeten zetten dat hij naar links stuurt als hij een muur aan de rechterkant merkt. Bij AI leert hij zelfstandig met een baan, wat veel flexibeler is. Op de baan past hij zich aan de omstandigheden, en kan hij wellicht wel dingen verbeteren waar wij niet eens aan gedacht hebben.

Om de AI een lichaam te kunnen geven gebruiken we een LIDAR-sensor voor de ogen, een computer voor het brein, en de microcontroller om signalen vanuit de computer naar de rc-auto te sturen.

Voor de computer gebruiken we de raspberry pi. De raspberry pi is een mini-computer die makkelijk op de auto past. De raspberry pi is goedkoop compact en nog redelijk snel.

Voor de microcontroller gebruiken we de arduino. We gebruiken de arduino uno omdat het een van de meest gedocumenteerde en populaire microcontrollers op de markt is.

Voor de auto gebruiken we een himoto bad boy. De Himoto bad boy heeft loskoppelbare draadjes bij de receiver die we dan rechtstreeks in de arduino kunnen doen. Een auto zonder deze functie zou flink wat knippen en plakken nodig hebben gehad. Ook is de himoto auto een zeer accurate auto volgens bronnen online, zodat de auto niet sneller of slomer gaat dan de AI bedoeld heeft. Dat zou ons een hoop frustratie en hoofdpijn geven.

Voor de code gebruiken we de programmeertaal python. Python is het meest gebruikt en meest ontwikkeld in het veld van AI.

H3 - Kunstmatige intelligentie

Aangezien bij wegen veel verschillende bochten, wegbreedte en obstakels mogelijk zijn, is het onmogelijk om voor alle situaties regels te programmeren. Hierdoor moet een computer in staat zijn om eigen beslissen te maken op de weg, hiervoor gebruiken we kunstmatige intelligentie.

3.1 Neuraal netwerk

Kunstmatige intelligente verwijst vaak naar neurale netwerken, een neuraal netwerk probeert het principe van het menselijk brein toe te passen op computers.

3.1A Hoe werkt een neuraal netwerk?

Een neuraal netwerk is een wiskundig model dat vanuit een input een output kan genereren. Een neuraal netwerk heeft een input, hidden en output laag. In de input laag gaat alle informatie, de informatie wordt in de hidden en output laag verwerkt en de resultaten van de output laag is ook het resultaat van een neuraal netwerk. In figuur 1 zie je een neuraal netwerk met de verschillende lagen.

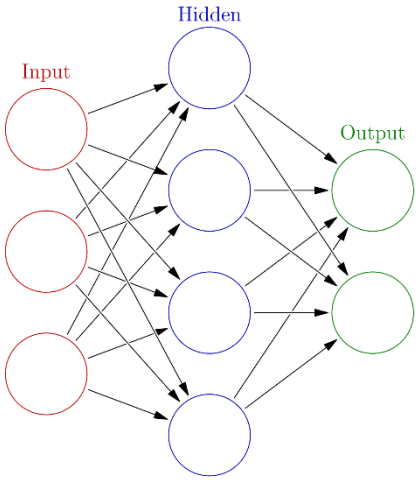
In figuur 1 zijn de bolletjes neuronen. Elke neuron heeft een activatiewaarde van 0 tot 1 of -1 tot 1, bij ons heeft elke neuron een waarde van -1 tot 1. De input neuronen worden geactiveerd door de informatie wat in het model wordt gevoerd. In de hidden en output neuronen wordt de wiskunde gedaan. Uiteindelijk krijgt elke output neuron een activatiewaarde -1 tot 1, dit is de output van het neuraal netwerk.

3.1B Wat gebeurt er in een neuraal netwerk?

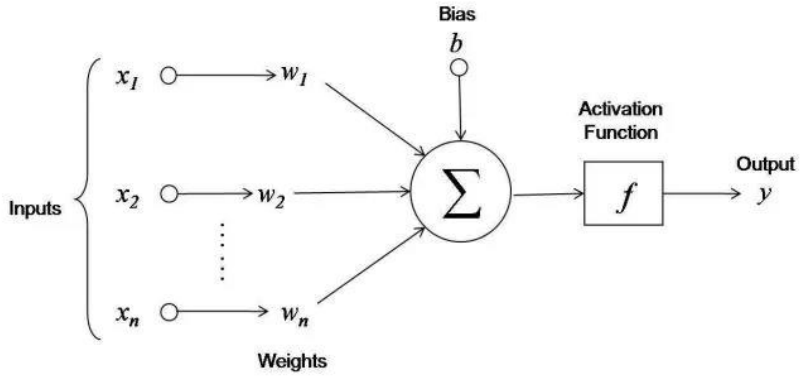
Elke neuron in deze lagen is een functie die een activatiegetal y geeft. In figuur 2 zien we wat er gebeurt in een individueel neuron in de input of output laag. Eerst worden alle inputs x die verbonden zijn aan het neuron vermenigvuldigd met een bijbehorend gewicht w . Hoe belangrijker de verbinding tussen de input neuron en het neuron, hoe hoger het gewicht. Het neuron heeft ook een bias b , dit bepaalt hoe belangrijk deze neuron is. Als we de som van alle inputs keer gewichten en de bias nemen krijgen we een opsomming Σ . Ten slotte moet het activatiegetal van elke neuron een waarde tussen -1 en 1 hebben, dus Σ wordt in een tanh activatiefunctie σ gestopt. Elke neuron heeft de formule:

$$y = \sigma(w_1x_1 + w_2x_2 + w_nx_n - b).$$

Deze formule zit in elke neuron in de hidden en output laag, met verschillende gewichten en bias. De gewichten en bias van elke neuron zijn dus parameters die je kan veranderen om het neuraal netwerk beter te laten presteren (Arthur Arnx, 2019), meer hierover in 3.2.



Figuur 1 algemene structuur neuraal netwerk



Figuur 2 individuele neuron

3.1C Wat is de input/output van ons neuraal netwerk?

Het neuraal netwerk dat wij gebruiken om de auto zelf te laten rijden werkt precies hetzelfde als de neuraal netwerken, maar het is wel belangrijk om te begrijpen welke input data ons neuraal netwerk nodig heeft en wat voor output data ons neuraal netwerk geeft.

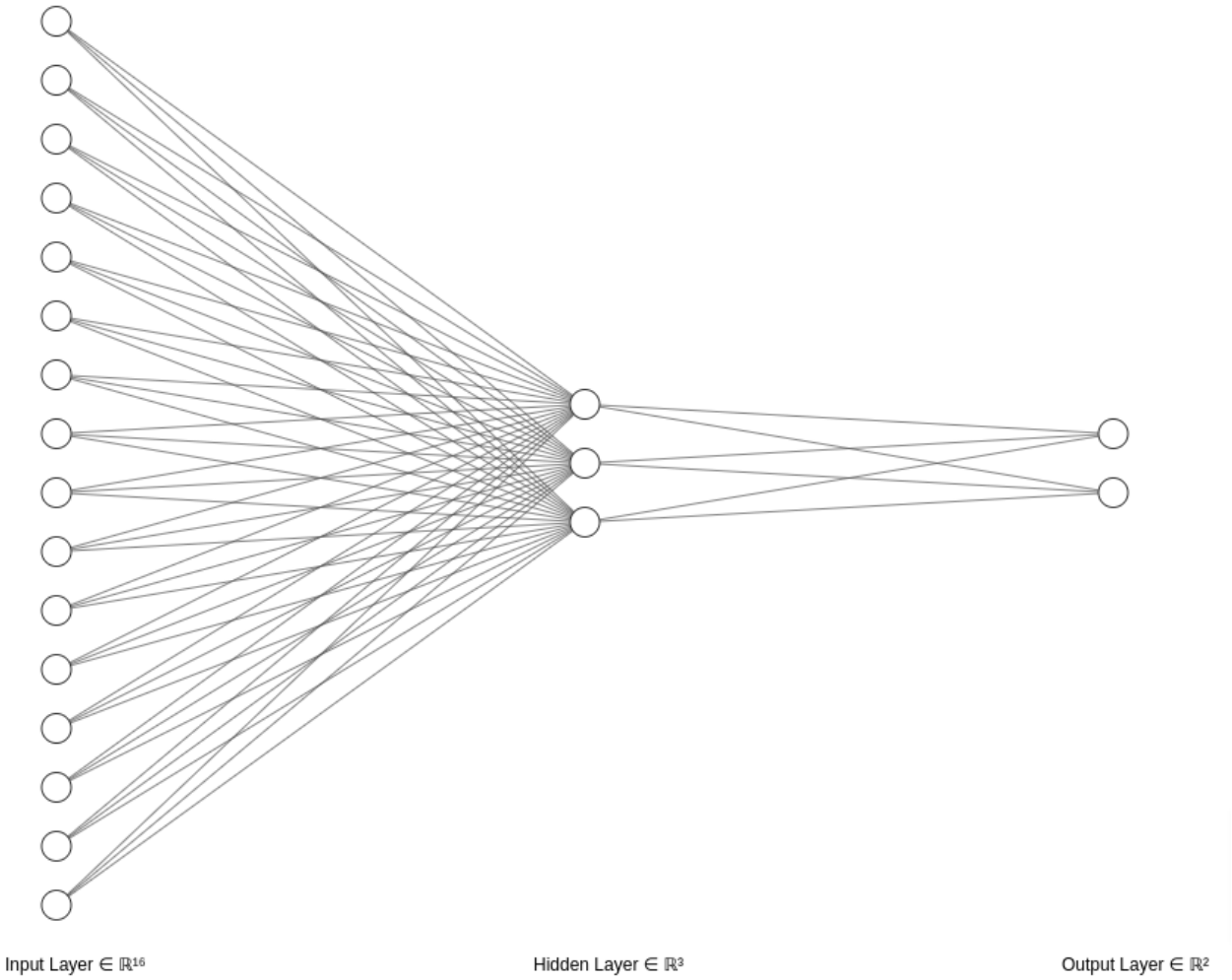
Als input data krijgt ons neuraal netwerk afstandspunten van een LiDAR sensor, dit is een sensor dat de afstand geeft van de LiDAR en een muur of obstakel. Deze informatie van de LiDAR sensor gebruikt het neuraal netwerk als input.

In de output laag zijn er twee neuronen, een voor sturen en een voor gas. Zoals we weten draagt elke neuron een waarde tussen -1 en 1. Bij het stuur neuron betekent –1 volledig naar links en 1 volledig naar rechts. Bij het gas neuron betekent –1 gas los en 1 vol gas.

3.1D Hoe ziet ons neuraal netwerk eruit?

Het beste model dat wij hebben getraind heet de Daniel v3. Het model heet zo omdat we in 2024 helaas afscheid moesten nemen van onze goede vriend Daniël van der Maas, die naar Nieuw-Zeeland is verhuisd.

We zien dat Daniel v3 is geëvolueerd om 3 hidden neuronen te hebben. Opmerkelijk is dat er maar 3 aantal neuronen nodig waren om een gesofisticeerd model te trainen dat in nieuwe situaties snel kan aanpassen, hoe we hebben getraind komt terug in 4.2.



Figuur 3. De netwerk structuur van Daniel v3

3.2 Machine learning

Machine learning is een vak dat zich richt op het trainen van AI-modellen. Het trainen van AI-modellen is het veranderen van de parameters totdat er een gewenst model ontstaat. Er zijn verschillende soorten machine learning algoritmes om dit doel te bereiken. Op dit hoofdstuk richten we op het NEAT machine learning algoritme.

3.2A Waarom NEAT?

Een veelgebruikte machine learning algoritme is supervised machine learning, hierbij wordt veel gelabelde data in een neuraal netwerk gevoerd waardoor het neuraal netwerk zelf patronen leert herkennen en voorspellen. Omdat wegen onvoorspelbaar zijn met verschillende bochtstraal, wegbreedte en obstakels. Is het onmogelijk om gelabelde data te vinden wat wij kunnen gebruiken. We moeten dus een zelf lerende machine learning algoritme hebben die neuraal netwerken evolueert op basis van hun prestaties. NEAT is hiervoor een geschikte kandidaat.

3.2B NEAT machine learning algoritme

NEAT staat voor Neuro-Evolution of Augmenting Topologies. NEAT wordt vaak gebruikt om hele specifieke problemen op te lossen waarvoor vaak geen training data beschikbaar is. Hoe beter een neuraal netwerk het probleem op lost, hoe hogere fitness waarde het neuraal netwerk krijgt. Met NEAT kan je neuraal netwerken laten evolueren totdat je het gewenste model creëert. De volgende sub paragrafen gaan over de theorie achter NEAT.

3.2C Het concept van NEAT (Theoretisch kader)

NEAT begint met een populatie eenvoudige neuraal netwerkstructuren, dan is vaak de input laag direct verbonden aan de output laag en zijn er geen hidden lagen. Met een fitness functie wordt de prestatie van een individueel genoom (neuraal netwerk) bepaalt. De genomen met de hoogste fitness worden door NEAT gekozen om te reproduceren. Hierdoor bestaat de volgende generatie uit genomen met mutaties of cross-overs van de best presterende modellen uit de vorige generatie. Dit proces herhaalt zich totdat de gewenste fitness waarde is bereikt (Trevor Burton-McCreadie, 2024).

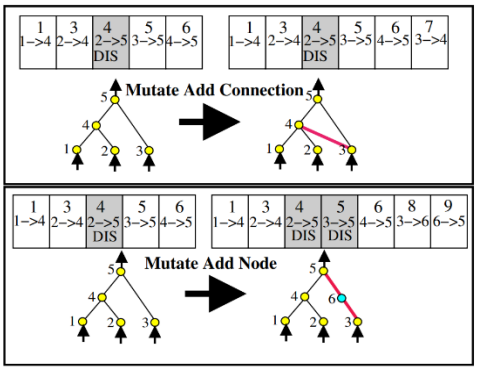
3.2D Mutatie NEAT (Theoretisch kader)

Om innovatie in de populatie te creëren worden genomen gemuteerd, dit is geïnspireerd door mutatie dat plaats vindt in de biologie. Mutatie komt voor in drie vormen voor bij NEAT, namelijk:

- Verbindingen maken tussen twee losse neuronen

- Een neuron toevoegen tussen twee verbonden neuronen, zo maakt NEAT zijn eigen hidden lagen
- Gewicht van een verbinding tussen twee neuronen veranderen

Het is logisch dat dit soort willekeurige mutaties vaak de fitness juist negatief zullen beïnvloeden, daarom worden de impact van deze mutaties geminimaliseerd. Zo beginnen nieuwe connecties tussen neuronen vaak met een gewicht van 0 (Robert MacWha, 2021). In figuur 4 zie je een visualisatie van hoe een genoom kan muteren.

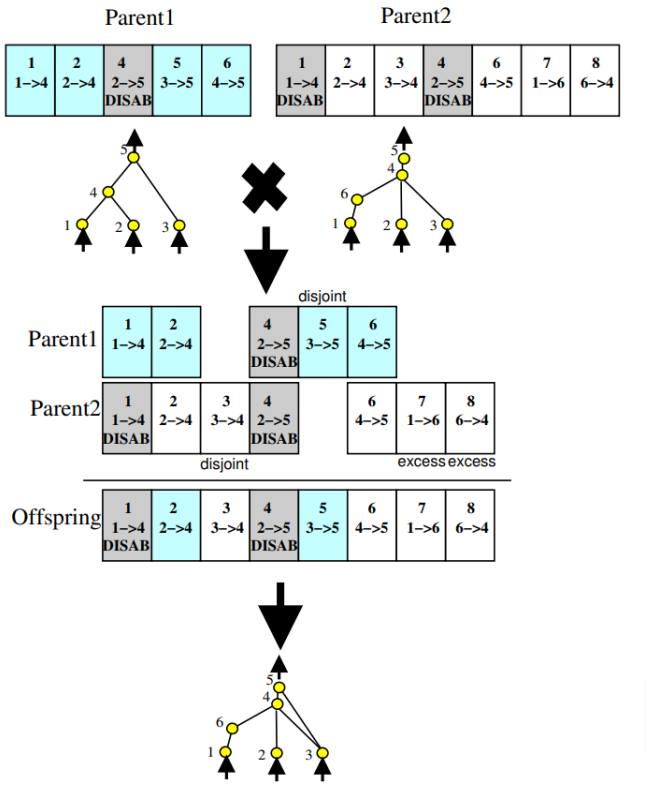


Figuur 4 mutatie van een genoom

3.2E Cross-overs NEAT (Theoretisch kader)

Een andere manier om nakomelingen te creëren kan NEAT twee goed presterende genomen combineren. Maar vaak verschillen de structuren van genomen, zo kan een genotype bijvoorbeeld 3 hidden nodes hebben en een ander 2.

NEAT lost dit op door ten eerste een dominant genoom te kiezen, vervolgens krijgt het kind een willekeurige verbinding van de verbindingen die beide ouders delen. Ten slotte neemt het kind de verbindingen van het dominante genoom over van de verbindingen die niet gedeeld worden door de ouders (Robert MacWha, 2021). In figuur 5 zie je hoe een nieuw genoom kan vormen uit cross-over (Stanley & Miikkulainen, 2002). Hierbij zien we ook de genen van een neuraal netwerk, genen representeren verbindingen tussen twee neuronen.



Figuur 5 cross-over van twee genomen

3.2F Speciation NEAT (theoretisch kader)

Mutatie in NEAT zorgt er vaak voor dat de fitness van een gemuteerd genoom achteruitgaat. Maar om voorruitgang te maken moeten genomen muteren, om dit probleem op te lossen wordt er gebruik gemaakt van speciation. Speciation splitst de populatie op in verschillende teams, zo wordt niet elke individuele genoom geëvalueerd en heeft elke unieke mutatie meer tijd om een hogere fitness te halen.

Hoeveel twee genomen op elkaar lijken wordt bepaald door een compatibiliteitswaarde waarbij:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W}$$

Hierbij is E het aantal genen in overmaat, D het aantal onsamenhangende genen en $\overline{W}$ het gemiddelde verschil van de passende genen. De coëfficiënten c worden gebruikt om de belangrijkheid van elke factor te veranderen. N is het aantal genen van het grootste genoom (Stanley & Miikkulainen, 2002).

Door de compatibiliteitswaarde van elke genoom te berekenen kunnen we elke genoom een team geven.

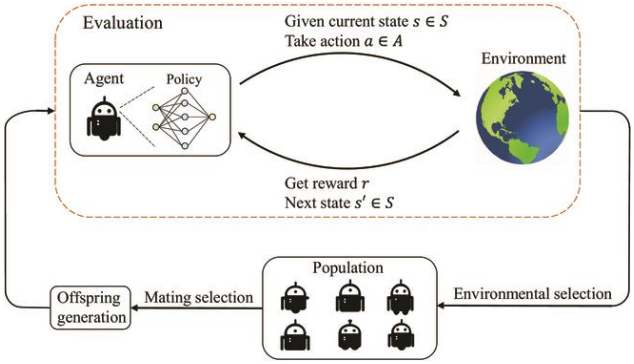
3.3 Toepassen machine learning

In deze paragraaf gaan we in op hoe we machine learning toepassen, we gaan het hebben over NEAT en de simulatie waarin we NEAT de AI laten evalueren en evolueren.

3.3A Hoe trainen we onze neuraal netwerken?

Om het model zelf te leren autorijden moeten we het trainen, hebben we gekozen voor het NEAT machine learning algoritme. We maken ook een eigen simulatieomgeving waar NEAT elke genoom kan evalueren met de fitness functie.

In figuur 6 zien we wat er gebeurt per generatie modellen. In het oranje vlak zien we hoe een individueel model wordt geëvalueerd op basis van zijn prestatie in de simulatie. Als alle modellen in één generatie zijn geweest gaan de beste modellen reproduceren en muteren zoals beschreven in de paragrafen over NEAT. Dit zorgt er in theorie voor dat met elke generatie onze gemiddelde fitness een beetje toeneemt.



Figuur 6. Flowchart van hoe NEAT modellen evalueert en evolueert

3.3B Wat is onze fitness functie?

Elk model wordt door NEAT geëvalueerd op basis van zijn fitness, een hogere fitness betekent dat het model een grotere kans heeft om te reproduceren met een ander model. Hoe we deze fitness evalueren moeten we zelf bepalen. Wij hebben ervoor gekozen om de fitness te evalueren op basis van de afgelegde afstand op de racebaan. Hoe groter de afstand een model kan afleggen, hoe hogere fitness waarde hij krijgt. Als een model een heel rondje kan rijden om het circuit krijgt het extra fitness punten. Het model kan ook gestraft worden, als een model botst tegen een muur krijgt het model juist een lagere fitness waarde. Het doel van het straffen en belonen is zodat de latere generaties modellen weten hoe ze moeten autorijden zonder te botsen tegen muren.

H4 Simulatie

Om onze neuraal netwerken te trainen hebben we een virtuele simulatieomgeving gebouwd in Python. De simulatie heeft realistische stuur en acceleratie modellen. In deze simulatieomgeving kunnen we de populatie modellen zelf rondjes laten rijden op een racebaan. Elke model krijgt zijn eigen auto, op deze auto zit ook een simulatie van een LiDAR sensor, die we ook zelf hebben geprogrammeerd. We trainen de modellen op een grote variatie racebanen.

Elke model moet zo ver mogelijk komen in onze simulatie, hoe verder een auto komt, hoe groter de fitness waarde zoals besproken in hoofdstuk 3. Nadat elke model is geweest in een populatie evalueert NEAT de modellen. Hierna kan een nieuwe populatie weer aan de slag. Zodra de simulatie een aantal aangegeven generaties heeft getraind stopt de simulatie automatisch en wordt de best presterende individuele model opgeslagen. Deze best presterende model kunnen we gebruiken in onze echte auto.

Onze voorspelling voor de simulatie is dat als we het een lange tijd laten lopen, de gemiddelde fitness van de gehele populatie groter is dan de eerste generatie. In figuur 6 hierboven zie je een flowchart van hoe de simulatie en NEAT samen zullen werken.

4.1 Structuur van de code

4.1A Klassen

De code van onze simulatie is opgebouwd uit verschillende klassen voor de verschillende taken die de auto moet verrichten. De hoofdklasse heet bijna altijd, en ook in ons geval, “Main”. In onze main code staat de hoofdcode. Het is de plek waar alle andere klasse bij elkaar komen om te functioneren. We hebben twee klassen voor de auto. 1 voor de LIDAR-sensor, en de andere voor de logica van de auto. De twee klassen worden hieronder uitgelegd.

4.1B “Car” klasse

In deze klasse wordt alle logica van de auto behandeld. We hebben een car.py bestand, die we importeren in het main.py bestand. In car.py hebben we een klasse die Car heet:

```
class Car:
```

Om de Car klasse te initiëren, moeten we eerst een initialisatie functie gebruiken, ook wel init functie genoemt voor gemak:

Car.py Python

```
def __init__(self, WIDTH, HEIGHT, track_init, car_img, checkpoint):
```

Deze initialisatie functie heeft een aantal parameters, zoals:

- WIDTH, De van de window waarin de simulatie zich afspeelt.
- HEIGHT, De hoogte van de window waarin de simulatie zich afspeelt.
- track_init, een lijst met waardes van het circuit, zoals: startpunt, checkpoints (checkpoints voeren we later zelfstandig in), en waar het bestand van het circuit is opgeslagen.
- Car_img, de locatie van de foto van de auto die we gebruiken
- Checkpoint, het checkpoint van het circuit, zodat die weet wanneer hij de startlijn als finishlijn moet zien als hij dit punt passeert.

Car.py Python

```python
def __init__(self, WIDTH, HEIGHT, track_init, car_img, checkpoint):
        self.checkpoint = checkpoint
        self.car_img = pygame.transform.scale(car_img, (30, 20))
        self.car_rect = car_img.get_rect(center=(WIDTH//2,
HEIGHT//2))  # Car's starting position
        # Constants for RC car behavior
        self.MAX_VELOCITY = 8  # Max speed is higher for an RC car
        self.ACCELERATION_RATE = 0.2  # RC cars accelerate faster
        self.BRAKE_RATE = 0.1  # RC cars decelerate faster
          # RC cars can turn more sharply
        self.TURN_RATE = 8  # Faster turn rate
        self.FRICTION = 0.95  # Reduced friction to simulate more
skidding
        self.velocity = 0  # Initial velocity
        self.angle = 0  # Initial orientation (angle in radians)
        self.angular_velocity = 0  # Initial angular velocity
        self.acceleration = 0  # Acceleration
        self.steering_angle = 0  # Steering input
        self.car_length = 30
        self.distance_covered = 0
        self.track_init = track_init
        self.checkpoint_reached = False
        self.x, self.y = track_init  # Initial position
```

In de init functie zet een aantal constantes neer, deze constantes zijn gehaald uit de parameters die we ingevoerd hebben en andere vaste waardes, zoals MAX_VELOCITY, oftewel de maximale snelheid, deze waardes spreken voor zich.

Daarnaast hebben we de stuurfunctie:

Car.py Python

```python
    def steering(self, max_steering_angle):
        self.max_steering_angle = max_steering_angle

        if max_steering_angle > 0:
            self.steering_angle = min(self.steering_angle + 2, self.max_steering_angle * 70)

        elif max_steering_angle < 0:
            self.steering_angle = max(self.steering_angle - 2, self.max_steering_angle * 70)

        else:
            self.steering_angle *= 0.9  # Return steering gradually to center
```

De max_steering_angle is de stuur input die de AI geeft aan onze auto functie, in deze functie draait die het stuur langzaam totdat die de bedoelde stuurhoek heeft, en als er geen stuurinput wordt gegeven gaat het stuur weer recht.

Handling, oftewel het gas geven en remmen:

Car.py Python

```python
def handling(self, throttle):
        self.throttle = throttle
        if self.throttle > 0:
            self.acceleration = self.throttle * 0.2
        elif self.throttle < 0:
            self.acceleration = self.throttle * 0.1
        else:
            self.acceleration = 0
```

Dit is precies hetzelfde concept als het sturen, maar dan met versnelling.

In change_velocity wordt de snelheid aangepast op basis van het gasgeven van de vorige functie en frictie

Car.py Python

```python
    def change_velocity(self):
        self.velocity += self.acceleration
        self.velocity = max(min(self.velocity, self.throttle * self.MAX_VELOCITY), 0)
        self.velocity *= self.FRICTION
```

Hier past hij de versnelling aan totdat de versnelling hoger is dan de maximale snelheid. Daarna past het model wrijving toe.

Hier wordt het sturen toegepast, we gebruiken de Ackermann stuur formule, omdat simpelweg het draaien van het auto model met x graden om zijn as niet realistisch is.

Car.py — Python

```python
    def moving_car(self):
        # If there's velocity, calculate the turning radius
        if self.velocity != 0 and self.steering_angle != 0:
            turning_radius = self.car_length /
math.sin(math.radians(abs(self.steering_angle)))
            self.angular_velocity = self.velocity / turning_radius

            # Update angle based on steering and velocity
            if self.steering_angle < 0:
                self.angle -= math.degrees(self.angular_velocity) * (self.velocity /
self.MAX_VELOCITY)
            else:
                self.angle += math.degrees(self.angular_velocity) * (self.velocity /
self.MAX_VELOCITY)
        else:
            self.angular_velocity = 0

        # Calculate new position
        self.x += self.velocity * math.cos(math.radians(self.angle))
        self.y -= self.velocity * math.sin(math.radians(self.angle))  # Y-axis is inverted in
Pygame
        # Update car rect position

        self.car_rect.center = (self.x, self.y)
        # Rotate car image
        rotated_car = pygame.transform.rotate(self.car_img, self.angle)  # Pygame uses
clockwise rotation
        rotated_rect = rotated_car.get_rect(center=self.car_rect.center)
        return rotated_car, rotated_rect
```

Hoewel we niet precies begrijpen waarom het Ackermann stuur formule werkt, werken auto’s in het echt ook met het Ackermann stuursysteem.

De volgende paar functies groeperen we even omdat ze niet zo uitgebreid zijn:

Car.py Python

```
    def report_position(self):
        return self.x, self.y, self.angle
    def get_velocity(self):
        return self.velocity
    def is_moving(self, start_time):
        if self.velocity == 0 and (time.time() - start_time) >= 1:
            return False
        else:
            return True
```

De eerste functie, report_position, wordt gebruikt om terug te koppelen aan het programma wat de positie van de auto is. We hebben dat nodig zodat de gesimuleerde lidar weet vanaf waar hij stralen moet sturen. De tweede functie, get_velocity, gebruiken we om de snelheid van de auto in het main programma te weten, dit is voor ons handig om problemen te diagnosen. Bijvoorbeeld om te zien of de auto nou stil staat, of extreem sloom gaat. De laatste functie, is_moving, wordt gebruikt om de auto te vermoorden als de snelheid nul is, en de auto dus stil staat. We moeten dit wel na een bepaalde tijd doen omdat die anders geen kans heeft om op snelheid te komen. In deze code snippet is het dus 1 seconde.

De laatste drie functies:

Car.py Python

```
def reached_checkpoint(self):
        if ((self.checkpoint[0] - 20)< self.x < (self.checkpoint[0] + 20) and self.y <
self.checkpoint[1] + 40):
            self.checkpoint_reached = True
    def has_finished(self):
        self.reached_checkpoint()
        if  (self.track_init[0] - 20)< self.x < (self.track_init[0] + 20) and self.y >
self.track_init[1] - 40 and self.checkpoint_reached:
            return True
        else:
            return False
    def get_distance_covered(self):
        self.distance_covered = self.distance_covered + self.velocity
        return self.distance_covered
```

Reached_checkpoint kijkt of de auto al over het aangegeven checkpoint heen is gegaan. Als de auto over dit checkpoint heen is gegaan wordt de startlijn als finishlijn aangegeven.

Het model checkt of de auto over de finishlijn heen is gegaan met has_finished. De get_distance_covered functie meet hoe ver de auto is gekomen. Dit wordt gebruikt om het aantal ‘punten’ wat een auto krijgt te bepalen, in NEAT-terminologie wordt het fitness genoemd. Als een auto verder komt krijgt hij meer fitness. De auto’s met de hoogste fitness mogen verder voortplanten in ons eugenese model.

4.1C De lidar klasse

De lidar klasse is een iets simpelere en kleinere klasse dan de car klasse. De lidar zijn de zintuigen van de auto, bestaand uit 16 stralen die over een hoek van 180 graden aan de voorkant van de auto worden uitgestraald.

Net als de car klasse heeft de lidar klasse weer een init functie.

Car.py — Python

```python
class Lidar:
    def __init__(self, track, WIDTH, HEIGHT):
        self.NUM_RAYS = 16  # Number of LiDAR rays
        self.FOV = 360  # Field of view in degrees (360 for full circle)
        self.MAX_RANGE = 200  # Max range of LiDAR in pixels
        self.LIDAR_ANGLE_STEP = self.FOV / self.NUM_RAYS  # Angular increment per ray
        self.track = track
        self.WIDTH = WIDTH
        self.HEIGHT = HEIGHT
```

Hier worden weer waardes ingesteld, waardes die doorgegeven worden van main, of waardes die al vast staan. Wat elk variabel hier betekent is redelijk vanzelfsprekend.

De eerste functie van de lidar klasse is cast_ray:

Car.py Python

```
    def cast_ray(self, x, y, angle):
        for i in range(self.MAX_RANGE):
            ray_x = int(x + i * math.cos(angle))
            ray_y = int(y + i * math.sin(angle))

            # Check if the ray is out of bounds
            if ray_x < 0 or ray_x >= self.WIDTH or ray_y < 0 or ray_y >= self.HEIGHT:
                return (ray_x, ray_y), i

            # Check if the ray hits a black pixel on the track
            if self.track.get_at((ray_x, ray_y)) == (255, 255, 255, 255):  # Black pixel
(track)
                return (ray_x, ray_y), i

        # If no collision, return the maximum range
        return (int(x + self.MAX_RANGE * math.cos(angle)), int(y + self.MAX_RANGE *
math.sin(angle))), self.MAX_RANGE
```

in deze functie wordt er een laser straal gesimuleerd. De laserstraal wordt gestraald in de opgegeven directie, als de straal iets raakt, wordt de afstand van het geraakte punt tot de auto doorgestuurd, als de ray buiten het scherm gaat wordt dat aangegeven, en als de ray niets raakt wordt het verste punt doorgegeven.

In de simulate lidar wordt de cast_ray functie gebruikt om met kleine verhogingen steeds een lidar straal uit te zenden.

Car.py Python

```
    def simulate_lidar(self, x, y, car_orientation):
        distances = []
        rays = []

        for i in range(self.NUM_RAYS):
            angle = math.radians(car_orientation) + math.radians(i * self.LIDAR_ANGLE_STEP)  #
Convert angle to radians
            hit_point, distance = self.cast_ray(x, y, angle)
            distances.append(distance)
            rays.append(hit_point)


        return distances, rays
```

Er worden eerst twee lijsten gemaakt, afstanden, en stralen. Daarna wordt met de hoek van de auto en de eerder aangegeven hoek ten opzichte van de auto waarin je de straal wilt uitzenden, berekent welke kant de straal uitgezonden moet worden. Waarna de cast_ray functie wordt gebruikt, en de output van die functie in de lijst van distances gezet wordt.

4.1D het main bestand

Na alle klasse en functies daarbinnen kunnen we eindelijk alles bij elkaar laten komen bij het main bestand, het main bestand bestaat niet uit klassen, maar uit de functie eval_genomes. Dat is de functie die neat gebruikt om de AI te simuleren, daarnaast gebruiken we ook nog de pygame module binnen die functie om de auto te visualiseren.

D

main.py Python

```python
import pygame
import random
import pickle
import neat
import math
import time
import numpy as np
from car import Car
import pyautogui as pg
from lidar_sensor import Lidar
import matplotlib.pyplot as plt
import visualize
import os

os.chdir(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
pygame.init()
pygame.font.init()
my_font = pygame.font.SysFont('Comic Sans MS', 30)

WIDTH = 800
HEIGHT = 600
screen = pygame.display.set_mode((WIDTH, HEIGHT))
clock = pygame.time.Clock()
running = True


NUM_RAYS = 20  # Number of LiDAR rays
FOV = 360  # Field of view in degrees (360 for full circle)
MAX_RANGE = 200  # Max range of LiDAR in pixels
LIDAR_ANGLE_STEP = FOV / NUM_RAYS  # Angular increment per ray

tracks = [os.path.join("assets", "track1.png"), (420, 512), (413, 130)]
track = pygame.image.load(tracks[0]).convert()
car_img = pygame.image.load(os.path.join("assets", "yellow-car-top-view-free-
png.png")).convert_alpha()  # Smaller car to represent RC car
```

Hier worden alle benodigde modules geimporteerd. Daarna zetten we het hoofdpad op het pad waar main zich bevind met os.chdir(). Daarna initializeren we pygame, en zetten we het lettertype op comic sans omdat dat het beste lettertype is. Daarna zetten we de dimensies van het scherm, met WIDTH en HEIGHT. Daarna geven we de informatie over de locatie van het bestand van de track en de checkpoints. Andere benodigde assets worden daarna geladen.

Daarna komt de eval_genome functie.

main.py Python

```python
def eval_genomes(genomes, config):
    num = 0
    for genome_id, genome in genomes:
        num2 = 0
        num+=1
        net = neat.nn.FeedForwardNetwork.create(genome, config)
        car = Car(WIDTH, HEIGHT, tracks[1], car_img, tracks[2])
        lidar = Lidar(track, WIDTH, HEIGHT)
        running = True
        start = time.time()
        while running:

            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    running = False

            x, y, angle = car.report_position()
            if num2 == 0:
                distances, hit_points = lidar.simulate_lidar(x, y, -angle)
                normalized_distances = [d / MAX_RANGE for d in distances]
                output = net.activate(normalized_distances)
                steering = output[0]
                throttle = output[1]
                car.handling(throttle)
                car.change_velocity()
                car.steering(steering)

            check_if_hit, _ = lidar.simulate_lidar(x, y, -angle)
            for distance in check_if_hit:
                if distance <= 0:
                    running = False
            if not car.is_moving(start):
                running = False

            genome.fitness = car.get_distance_covered() / 1000.0
            rotated_car, rotated_rect = car.moving_car()
            screen.blit(track, (0, 0))
            screen.blit(rotated_car, rotated_rect.topleft)
            for hit_point in hit_points:
                if hit_points.index(hit_point) == 4 or hit_points.index(hit_point) == 12:

                    pygame.draw.line(screen, (0, 255, 0), (x, y), hit_point)
                else:
                    pygame.draw.line(screen, (255, 0, 0), (x, y), hit_point)

            if car.has_finished():
                genome.fitness += 5
                running = False
            # Update display
            #text_surface = my_font.render(str(round(x)) + "," + str(round(y)), False, (0, 0,
0))
            #screen.blit(text_surface, (0,100))
            text_surface = my_font.render(str(num), False, (0, 0, 0))
            screen.blit(text_surface, (0,0))
            fps_surface = my_font.render(str(round(clock.get_fps())), False, (0, 0, 0))
            screen.blit(fps_surface, (0,50))
            fps_surface = my_font.render(str(round(car.get_velocity())), False, (0, 0, 0))
            screen.blit(fps_surface, (0,100))
            fps_surface = my_font.render(str(time.time() - start), False, (0, 0, 0))
            screen.blit(fps_surface, (100,0))
            pygame.display.flip()
            num2 += 1
            if num2 == 11:
                num2 = 0
            # Frame rate
            clock.tick(5000)
```

Hier is waar alles gebeurt. Hier wordt de AI functie uitgevoerd, in de AI functie wordt pygame uitgevoerd. De vormen worden op het schrem getekend, en de lidar inputs worden aan de ai gegevne die ons een output, die we aan de auto doorgeven.

main.py Python

```python
def run_neat(config_file, checkpoint=None):
    config = neat.config.Config(
        neat.DefaultGenome,
        neat.DefaultReproduction,
        neat.DefaultSpeciesSet,
        neat.DefaultStagnation,
        config_file
    )

    # Create the population, which is the top-level object for a NEAT run

    if checkpoint:
        population = neat.Checkpointer.restore_checkpoint(checkpoint)
    else:
        population = neat.Population(config)

    # Add a reporter to show progress in the terminal
    population.add_reporter(neat.StdOutReporter(True))
    stats = neat.StatisticsReporter()
    population.add_reporter(stats)
    population.add_reporter(neat.Checkpointer(generation_interval=5,
filename_prefix='neat-checkpoint'))


    # Run for up to 50 generations
    winner = population.run(eval_genomes, 1)
    with open(os.path.join("neat", "best_genome.pkl"), 'wb') as f:
        pickle.dump(winner, f)
    # Display the winning genome
    print('\nBest genome:\n{!s}'.format(winner))
    pygame.quit()
    visualize.draw_net(config, winner, True) # plot best neural network
    visualize.plot_stats(stats, ylog=False, view=True) # plot average/best fitness'
```

Hier wordt neat uitgevoerd, er word teen configuratie file ingeladen, daarna wordt er een checkpoint geladen als die beschikbaar is. Daarna wordt er een reporter toegevoegd voor de statistieken. Nadat een generatie is gerund wordt het beste model opgeslagen met behulp van pickle. Daarna wordt pygame afgesloten en wordt er een grafiek getekend.

4.2 Realistisch simuleren

Hoe simuleren we de natuurkunde in onze python programma? In onze simulatie hebben we geprobeerd om zo een realistisch mogelijk milieu te maken voor onze auto. We hebben verschillende omwegen moeten maken om de auto realistisch kunnen te laten sturen, we hebben bijvoorbeeld de Ackermann steering geometry gebruikt om de auto zo realistisch mogelijk te draaien.

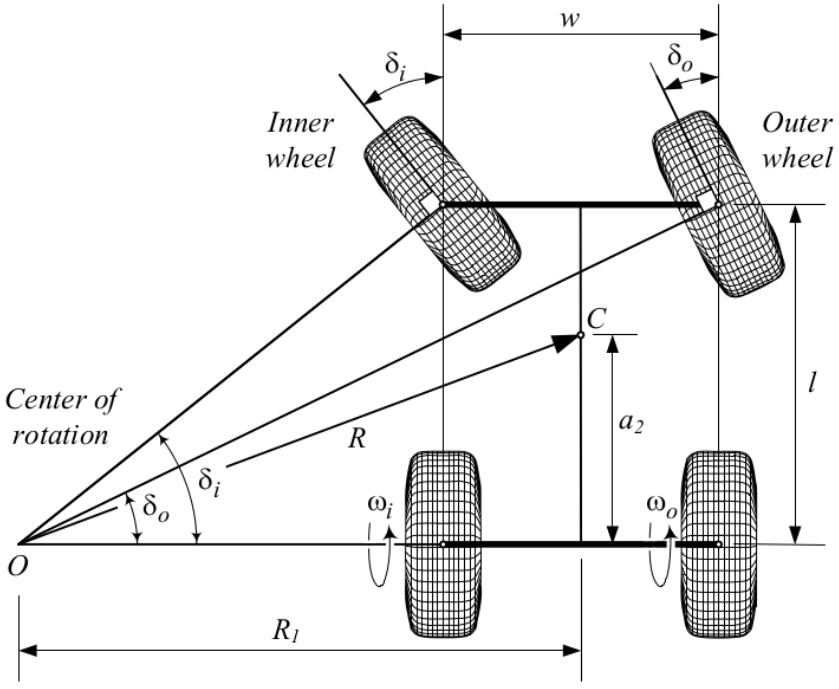
$$\tan(\theta) = \frac{L}{R - \frac{w}{2}}$$

Ackermann steering formule voor voorste wielen.

$$\tan(\theta) = \frac{L}{R + \frac{w}{2}}$$

Ackermann steering formule voor achterste wielen.

L is de afstand tussen de voor en achterwielen. W is de de afstand tussen de linker en rechter wielen, R is de draai radius.



Figuur 7. Ackermann stuur geometrie

H5 Ontwerp zelfrijdende auto

In dit hoofdstuk gaan we in hoe onze zelfrijdende auto werkt, we gaan elke component beschrijven en uitleggen hoe deze componenten met elkaar samenwerken om de auto te laten rijden.

5.1 Hoe werkt onze zelfrijdende auto?

Hier leggen we uit hoe onze zelfrijdende auto werkt. Elke 125 milliseconden stuurt de LiDAR sensor de afstandspunten naar de Raspberry Pi. De Raspberry Pi is het brein van onze auto, hierop staat namelijk het neuraal netwerk. Het neuraal netwerk gebruikt deze afstandspunten van de LiDAR als input en als output geeft het neuraal netwerk aan hoeveel de auto moet sturen en gas geven. Om te zeggen tegen de RC-auto hoeveel hij moet sturen en gassen gebruiken we een Arduino. De Arduino staat namelijk direct aangesloten aan de servomotoren van de RC-auto, hiermee kunnen we uiteindelijk de RC-auto besturen. Dit proces herhaalt zich elke 125 milliseconden. Het proces staat getekend in figuur 8.

Figuur 8. flowchart processen zelfrijdende auto

5.1A Wat is een API? (Theoretisch kader)

Een API is een application programming interface, het is een soort middelman tussen twee machines. API's zijn overal in onze wereld, als je iets op zoekt op google vraagt jouw apparaat voor data van de google servers, het vragen en ophalen van de data wordt geregeld via de google API. Min of meer elke website, app, programma in de wereld gebruikt API's om data aan de gebruiker te weergeven.

In ons project gaan wij ook veel gebruik maken van API's, we gaan bijvoorbeeld de YDLidar Python API gebruiken om met onze LiDAR sensor te communiceren. Daarnaast gebruiken we ook de SSH (secure shell host) API om de Raspberry Pi te besturen met onze laptop. Ook gaan we met de Arduino API programmeren om de Arduino te laten communiceren met de RC-auto.

Om deze API's te gebruiken hebben we veel programmeerervaring nodig, zo is de YDLidar Python API alleen te gebruiken door Python code te schrijven. Terwijl de Arduino API een variant van C++ gebruikt.

5.2 LiDAR sensor

De LiDAR sensor zijn de ogen van ons ontwerp, LiDAR staat voor Laser **I**maging **D**etection **A**nd **R**anging (Wikipedia, 2024). De LiDAR sensor die wij gebruiken is de YDLidar X2. We hebben deze gekozen omdat hij licht is en 360 graden kan scannen. LiDAR sensoren worden veel gebruikt in zelfrijdende auto's.

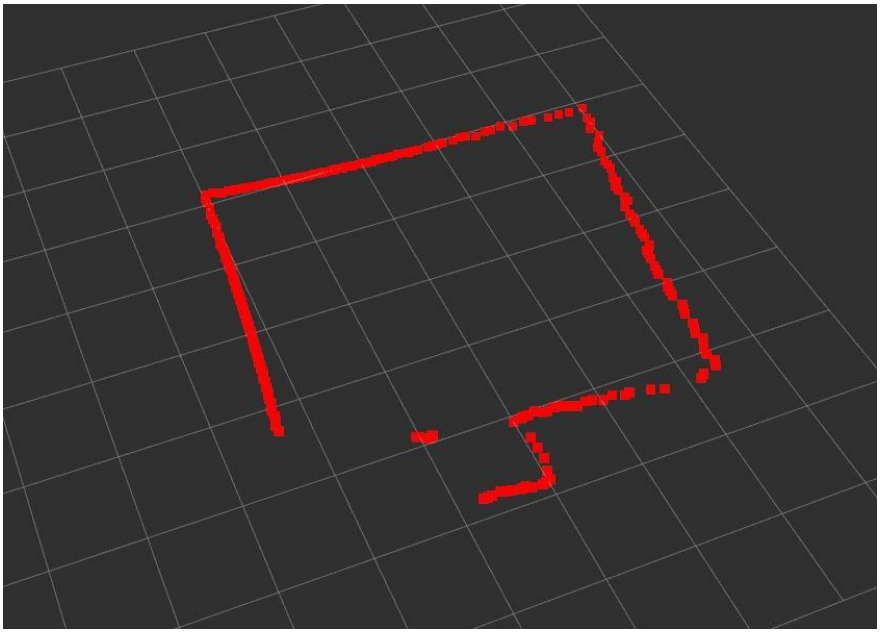
5.2A Hoe werkt de LiDAR sensor?

De LiDAR sensor werkt door een laserstraal uit te zenden. Wanneer deze laserstraal een object raakt, kaatst de laser terug naar de sensor. Omdat de laserstraal met lichtsnelheid gaat kunnen we met de tijd waarmee een laserstraal heen en weer gaat de afstand bepalen met:

$$s = \frac{t \cdot c}{2}.$$

Onze LiDAR sensor draait rond om per omwenteling een 360 graden scan van zijn omgeving te maken. Per rotatie krijgen we 375 afstandspunten. In figuur 9 zie je een LiDAR scan van een kamer.

De LiDAR en de raspberry Pi zijn direct aan elkaar verbonden via een USB-verbinding en met de YDLidar Python-API kunnen we direct de LiDAR informatie ontvangen en doorgeven aan het neuraal netwerk.



Figuur 9. LiDAR scan van een kamer

5.3 Raspberry Pi

Een raspberry pi is een compacte, goedkope en veelzijdige computer. Wij gebruiken de Raspberry Pi 3B, deze past makkelijk in onze RC-auto. De Raspberry Pi verbruikt niet veel elektriciteit, dit is ideaal voor ons omdat we de Raspberry pi hebben aangesloten aan een powerbank. De Raspberry Pi is onze centrale computer van de auto, de raspberry pi is verantwoordelijk voor het ontvangen en verwerken van de LiDAR data. Daarna moet de Raspberry Pi het verwerkte data in de juiste waardes sturen naar de Arduino.

5.3A Hoe verwerkt de Raspberry Pi de LiDAR afstandspunten?

We hebben ons model getraind op 16 afstandspunten met tussen elke afstandspunt een hoek van 22,5 graden. Maar onze LiDAR geeft ons 375 afstandspunten per rotatie, dit is te veel voor ons model. Daarom moeten we veel afstandspunten weg filtreren. In theorie is dit een makkelijk probleem om op te lossen, je kan de bijbehorende afstand van elke 22,5 graden die de LiDAR scant nemen. In de werkelijkheid is dat niet zo, onze LiDAR scant niet elke keer perfect dezelfde hoek tussen scans. Er zit altijd een beetje afwijking. Waardoor we dus de meest dichtstbijzijnde punt moeten nemen. Dit zorgt voor kleine afwijkingen van ongeveer 0,05 graden bij elke punt.

Als de LiDAR een volledige rotatie van ongeveer 360 graden heeft gemaakt, stuurt het een array aan afstandspunten in meters naar de Raspberry Pi. Er moet nog een ding gebeuren voordat het door de neuraal netwerk verwerkt kan worden, we moeten alle afstanden een waarde tussen 0 en 1 geven. We doen dit door alle afstanden te delen door het maximale scanbereik van de LiDAR.

Wanneer het neuraal netwerk heeft bepaald hoeveel het wilt sturen en gas geven, moeten we deze output data omzetten naar iets wat de RC-auto kan begrijpen. Uiteindelijk kunnen we deze waardes doorgeven aan de Arduino.

5.3C Versimpelde code op de Raspberry pi

Om te laten zien hoe ons python programma er ongeveer uitziet op de Raspberry pi, hebben we versimpelde versie gemaakt van het programma. Het programma doet de dingen die we in 5.3A en 5.3B hebben beschreven.

```python
import ydlidar
import neat
import serial

arduino = serial.Serial('/dev/ttyUSB0', 9600) # start serial verbinding met arduino
ydlidar.os_init() # start met lidar scannen
model = neat.Network.create("daniel_v4", "config.txt") # laad daniel_v4 (neuraal netwerk)

while ydlidar.os_isOk():
  scan_and_process()

def scan_and_process():
  scan = ydlidar.LaserScan()
  input_points = filter(scan) # filtreer de 16 afstandspunten
  normalized_input = [input_points / ydlidar.MaxRange()] # geef alle afstandspunten een waarde tussen -1 en 1
  network_output = net.activate(normalized_distances) # verwerk input
  steering = network_output[0]
  throttle = network_output[1]
  steering_degrees = (steering + 1) / 2 * 180 # convert tanh waarden naar graden voor servomotor
  throttle_degrees = (throttle + 1) / 2 * 180 # convert tanh waarden naar graden voor servomotor
  arduino.write(steering_degrees, throttle_degrees) # stuur signalen naar arduino
```

In de realiteit is het programma veel langer (100+ lijnen aan code).

5.4 GPIO

GPIO staat voor General-purpose input/output (Raspberry Pi, 2024). GPIO-kabels verbinden GPIO pinnen op verschillende elektronische apparaten. GPIO-kabels worden gebruikt om stroom, beaarding en signaal te vervoeren. Als de GPIO pinnen van meerdere elektronische apparaten zijn verbonden, is communicatie mogelijk tussen de meerdere elektronische apparaten.

5.4A Apparaten verbinden met GPIO

Om meerder apparaten te verbinden met GPIO moeten we de beaarding, stroom en signaal zelf verbinden. Hiervoor zijn er een aantal eisen.

- De signaal pin van de apparaten moeten verbonden zijn
- De stroom pin van de apparaten moet een spanningsbron hebben
- De beaarding van de spanningsbron en de apparaten moeten een gezamenlijke beaarding delen

5.4B Beaardingsboord

Een beaardingsboord is er zodat alle componenten in het circuit dezelfde referentiepunt voor de spanning heeft. Hierdoor worden de signalen van een apparaat veilig en fatsoenlijk doorgestuurd naar een ander apparaat. (Senetra, 2025)

5.5 Arduino aansluiten aan servomotoren

Om de RC-auto van afstand te bedienen met een afstandsbediening worden de signalen van de afstandsbediening ontvangen via de ontvanger. Deze ontvanger stuurt de juiste spanning en signalen naar de stuur en gas servomotoren en zo kan de RC-auto op afstand worden bediend. Door de servomotoren aan te sluiten aan de Arduino kunnen we de RC-auto met de Raspberry Pi besturen.

5.5A Hoe sluiten we de Arduino aan op de servomotor?

We hebben in totaal twee servomotor die individueel de gas en stuur van de RC-auto aandrijven. Een servomotor heeft drie GPIO pinnen voor stroom, beaarding en signaal.

De signaalpin van de Arduino en servomotor worden direct aan elkaar aangesloten.

De beaarding pin van de servomotor en Arduino worden aangesloten aan eenzelfde beaardingsbord.

De stroom pin van de servomotor wordt aangesloten aan de ontvanger, we sluiten de stroom aan op de ontvanger omdat de ontvanger al de juiste spanning levert van 6~8,4V (Himoto, z.d.) om de servomotor aan te drijven.

Omdat we nu ook de ontvanger gebruiken moet deze ook verbonden zijn aan de beaardingsbord waar de servomotoren en Arduino ook is aangesloten.

5.5B Hoe werkt een servomotor?

Een servomotor is een elektromotor die wordt gebruikt voor nauwkeurige positionering en snelheidsregeling. Het bestaat uit een motor, een feedbacksysteem zoals een encoder of potentiometer, en een regelsysteem dat de motor aanstuurt. Een invoersignaal, vaak via pulsbreedtemodulatie (PWM), bepaalt hoeveel de servomotor moet draaien. De feedbacksensor meet de huidige status en vergelijkt deze met de gewenste waarde. Bij afwijkingen stuurt het regelsysteem correctiesignalen naar de motor om de fout te minimaliseren. Dit proces herhaalt zich continu, waardoor de servomotor uiterst nauwkeurig werkt. Servomotoren worden veel gebruikt in robotica, industriële toepassingen, en RC-auto’s.

5.6 Arduino

Een Arduino is een compacte microcontroller dat geprogrammeerd kan worden in C++ om instructies te volgen. De instructies die wij aan de Arduino geven zijn hoeveel de Arduino de servomotoren moeten draaien voor gassen en sturen.

Zodra het neuraal netwerk op de Raspberry Pi heeft bepaald hoeveel het wilt gassen en sturen, worden de waarden van –1 tot 1 omgezet in graden van 0 tot 180. Dit zegt hoeveel de servomotoren moeten draaien om de auto te laten gassen en sturen (Arduino, 2024).

5.6A Hoe ontvangen we en sturen we signalen?

De Raspberry pi stuurt de gas en stuur instructies naar de Arduino via seriële verbinding (Roboticsbackend, z.d.). Seriële verbinding is een manier van informatie sturen tussen twee apparaten. Op de Arduino staat een programma dat de waarden tussen 0 en 180 ontvangt en die waarden gebruikt om de servomotoren aan te drijven. Hieronder zie je de pseudocode van wat er op de arduino staat, het programma doet wat net is beschreven.

```
void setup() {
    steeringServo.attach(9); // stuur motorservo staat op signaal pin 9
    throttleServo.attatch(10); // gas motorservo staat op signaal pin 10
}

void loop {
    if (Serial.available() > 0) {
      data = Serial.readString();
      steeringData = data.substring(0);
      throttleData = data.substring(1);
      steeringServo.write(steeringData); // hoeveel moet stuur servo draaien
      throttleServo.write(throttleData); // hoeveel moet gas servo draaien
    }
    else {
      throttleServo.write(90); // als er geen signaal meer komt vanuit de raspberry pi, dan stoppen met gas geven
    }
  }
```

H6 Testen (Werkwijze)

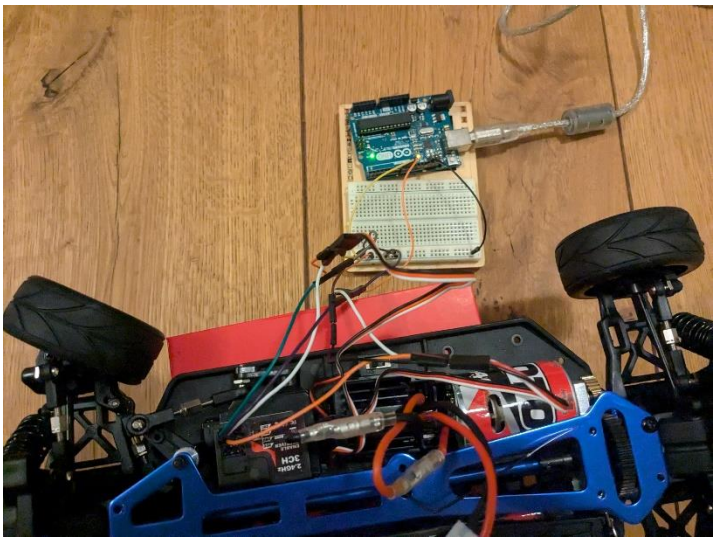
6.1 Hoe gaan we testen?

We testen veel verschillende componenten voordat we alles combineren om te testen of de auto zelf kan rijden. Hiervoor moeten we een aantal dingen testen:

1. Arduino + servomotoren
2. LiDAR + Raspberry Pi
3. LiDAR + Raspberry Pi + Arduino + servomotoren (statische test)
4. Zelfrijdende auto testen

6.1A Testen Arduino en servomotoren

We begonnen met het testen van de Arduino en servomotoren op de RC-auto. Dit deden we door de Arduino aan te sluiten op de servomotoren zoals beschreven in hoofdstuk 5.5. Hierna konden we met een computer via de Arduino de auto gas laten geven en sturen. In totaal duurde het aansluiten en testen 14 uur.

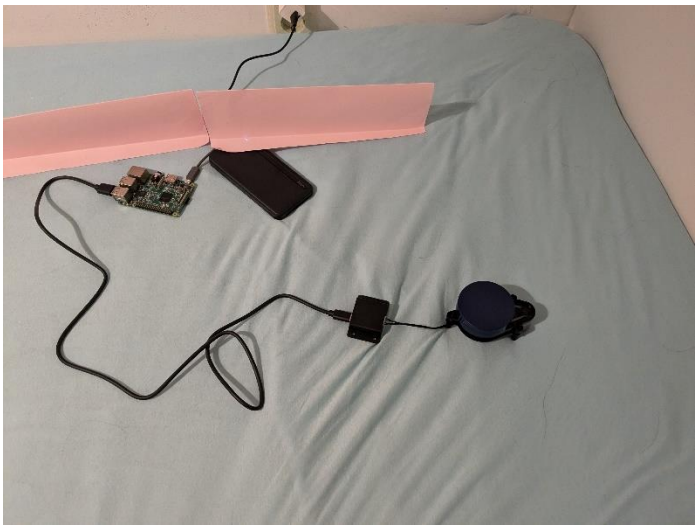


Figuur 10. Testen Arduino en servomotor

6.1B Testen LiDAR en Raspberry Pi

Om te kijken of Daniel v3 nog goede beslissen kon maken met input gegevens van een echte LiDAR sensor moesten we deze twee componenten samen testen. Tijdens het testen hadden we door dat de LiDAR veel verder kon scannen dan dat we hadden gesimuleerd en dus Daniel v3 slechte beslissingen maakte. Hierdoor moesten we een nieuwe versie van Daniel v3 trainen om het met de LiDAR compatibel te maken. Ook ontdekte we dat de

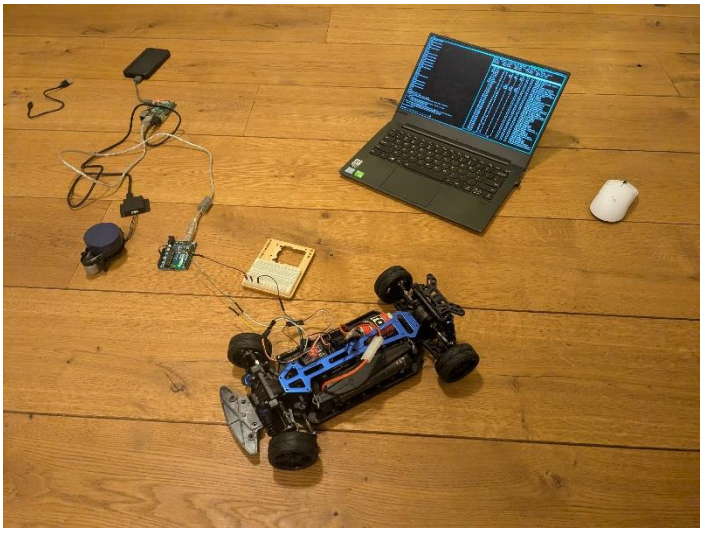
LiDAR niet perfect dezelfde hoek scande tussen elke afstandspunt (5.3A). Het testen duurde 5 uur.



Figuur 11. Testen Daniel v3 met LiDAR input

6.1C Testen LiDAR, Raspberry Pi, Arduino, servomotoren, RC auto

Om te kijken hoe alle componenten samenwerkte gingen we alle componenten samen testen op een platform. Hier lieten we de daniel v3 gas geven en sturen op basis van de LiDAR zoals staat in heel hoofdstuk 5. Het testen van alle delen samen duurde 3 uur.



Figuur 12. Testen alle onderdelen

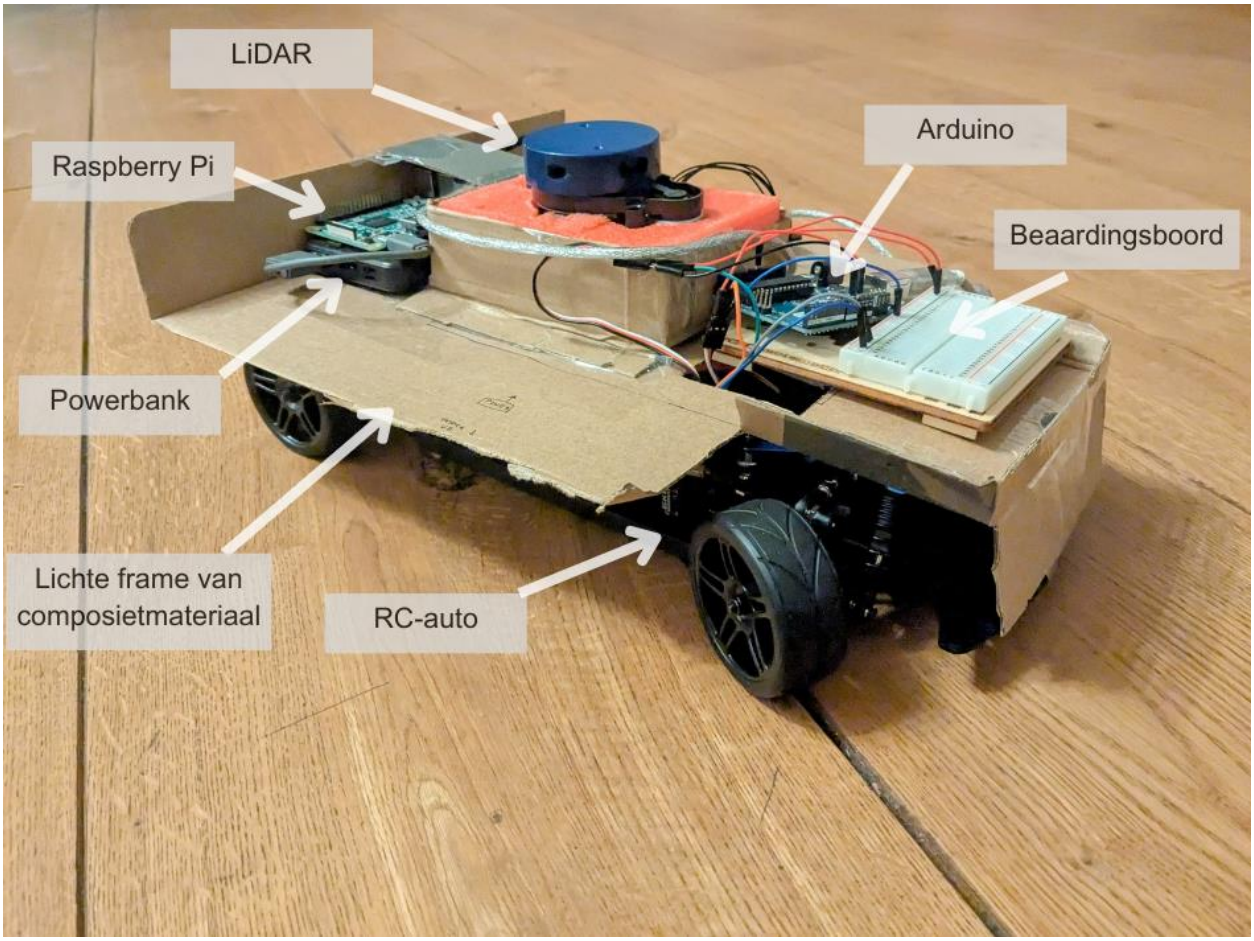
6.1D Testen zelfrijdende auto

Nadat de frame voor de componenten gebouwd was konden we de auto zelf laten rijden. Hiervoor hadden we een simpele racebaan gebouwd en probeerde we de auto een rondje te laten rijden. Het testen van de zelfrijdende auto duurde 12 uur.

Figuur 13. Testen zelfrijdende auto op racebaan

H7 Resultaten

In figuur 14 zie je de zelfrijdende auto, met alle componenten erop, de frame waarop de componenten staan is gemaakt van een veelzijdig composietmateriaal dat is vervaardigd uit meerdere lagen papierpulp. Dit materiaal is makkelijk te vervormen en is ontworpen voor structurele integriteit.



Figuur 14. Eindproduct zelfrijdende auto

7.1 Resultaten simulatie

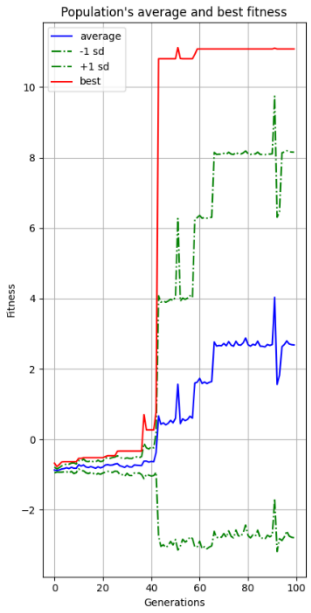
Deze paragraaf gaat over de resultaten van de simulatieomgeving waarin we met NEAT een populatie modellen hebben getraind en we gaan het hebben over het beste model uit deze simulatie.

7.1A Fitnesswaarden

In figuur 15 zien we de resultaten van de simulatie na 100 generaties op 4 verschillende banen. De rode lijn is de beste individuele fitness van een generatie. De blauwe lijn is de gemiddelde fitness van een generatie. De groene lijnen zijn de fitnesswaarden van de modellen die 1 standaardafwijking beter of slechter zijn van het gemiddelde.

We zien een aantal opmerkelijke dingen in deze grafiek:

1. De gemiddelde fitness neemt tot generatie 40 erg sloom toe
2. De fitness van de auto's die 1 standaardafwijking naar beneden afwijken van het gemiddelde gaat de min in
3. De beste fitness van elke generatie blijft lange tijden hetzelfde na generatie 40
4. De gemiddelde fitness neemt langzaam toe na generatie 40



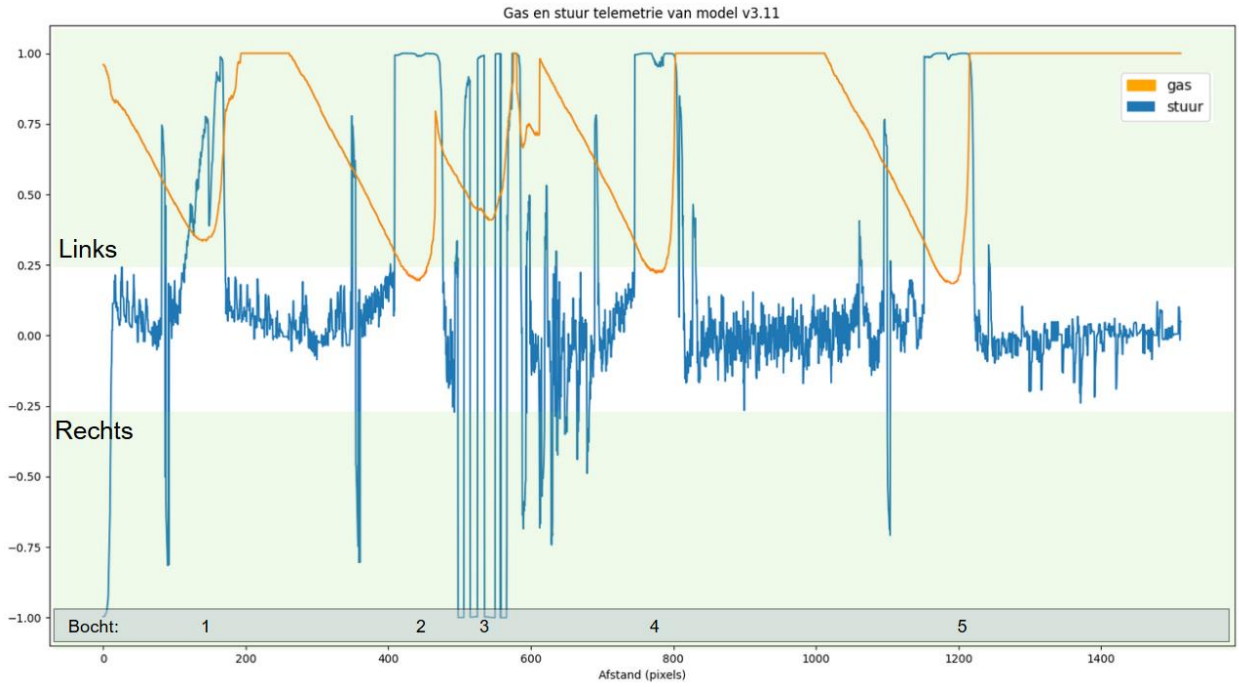
Figuur 15. Fitnesswaarden van de populatie aan de hand van generaties

7.1B Leert het beste model (Daniel v3) wel echt autorijden?

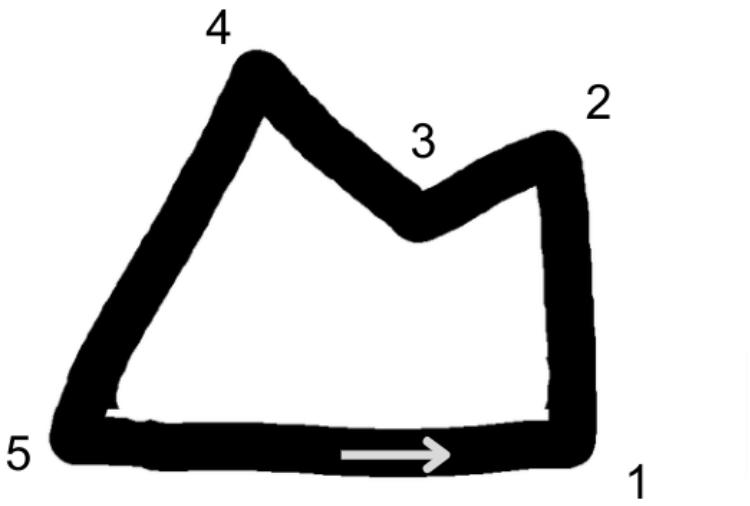
Om ervoor te zorgen dat een model niet alleen een hoge fitness waarden heeft omdat die de baan heeft “onthouden” testen we op veel verschillende banen. In figuur 16 zien we een baan dat Daniel v3 nooit heeft bereden.

In figuur 16 zien we de gas en stuur telemetrie van de auto. De telemetrie de output zien van het neuraal netwerk. De blauwe lijn is hoeveel hij stuurt, waarbij -1 volledig naar rechts en 1 volledig naar links is. De oranje lijn is hoeveel gas de auto geeft, waarbij 0 gas los en 1 vol gas is. De groene vlakken geven aan of de auto naar links of rechts gaat. Onderin zien we ook bij welke bocht de auto zich bevindt. We zien een aantal patronen:

1. De auto neemt voor elke bocht zijn gas los
2. Na elke bocht neemt de gas sterk toe
3. In de “apex” van elke bocht wordt er het minste gas gegeven
4. Bij het sturen van bocht 2, 4, 5 zijn er geen oscillaties
5. Bij het sturen van bocht 1 is een beetje oscillatie
6. Bij het sturen van bocht 3 is er veel oscillatie
7. In de rechte stukken oscilleert het sturen tussen –0,25 en 0,25. De auto blijft wel recht rijden.
8. Voor de bochten 1, 2, 4 en 5 stuurt de auto snel naar links en naar rechts



Figuur 16. Gas en stuur telemetrie van Daniel v3 op een baan dat het nooit heeft bereden



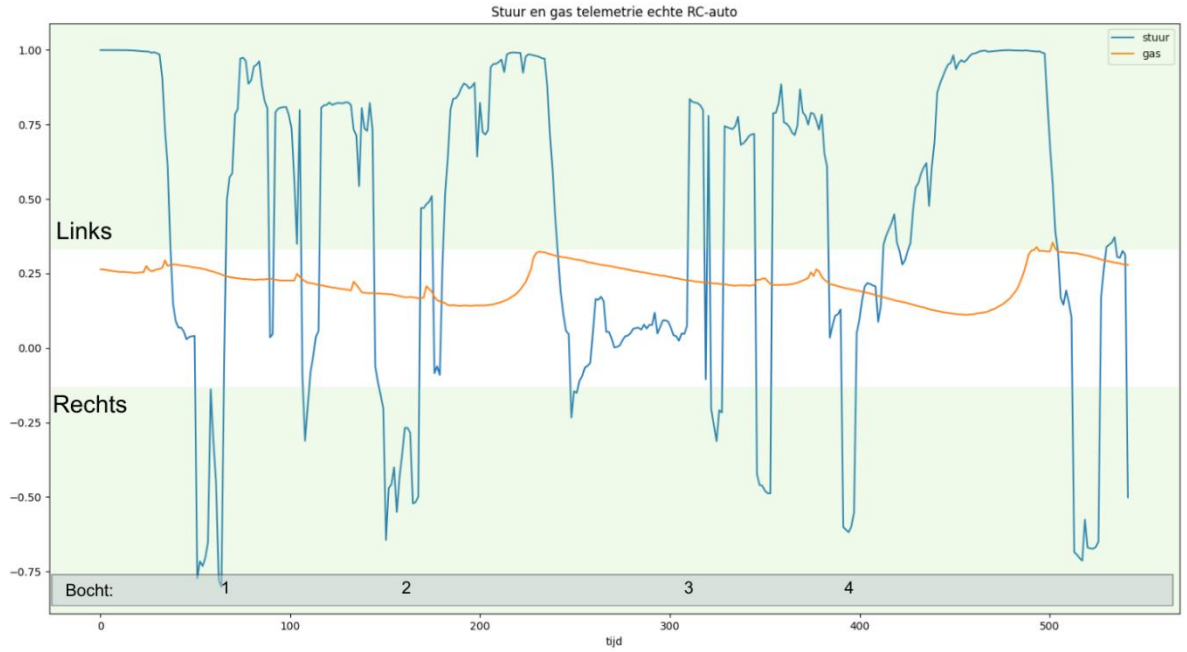
Figuur 17. Racebaan dat Daniel v3 nooit heet bereden

7.2 Resultaten zelfrijdende auto

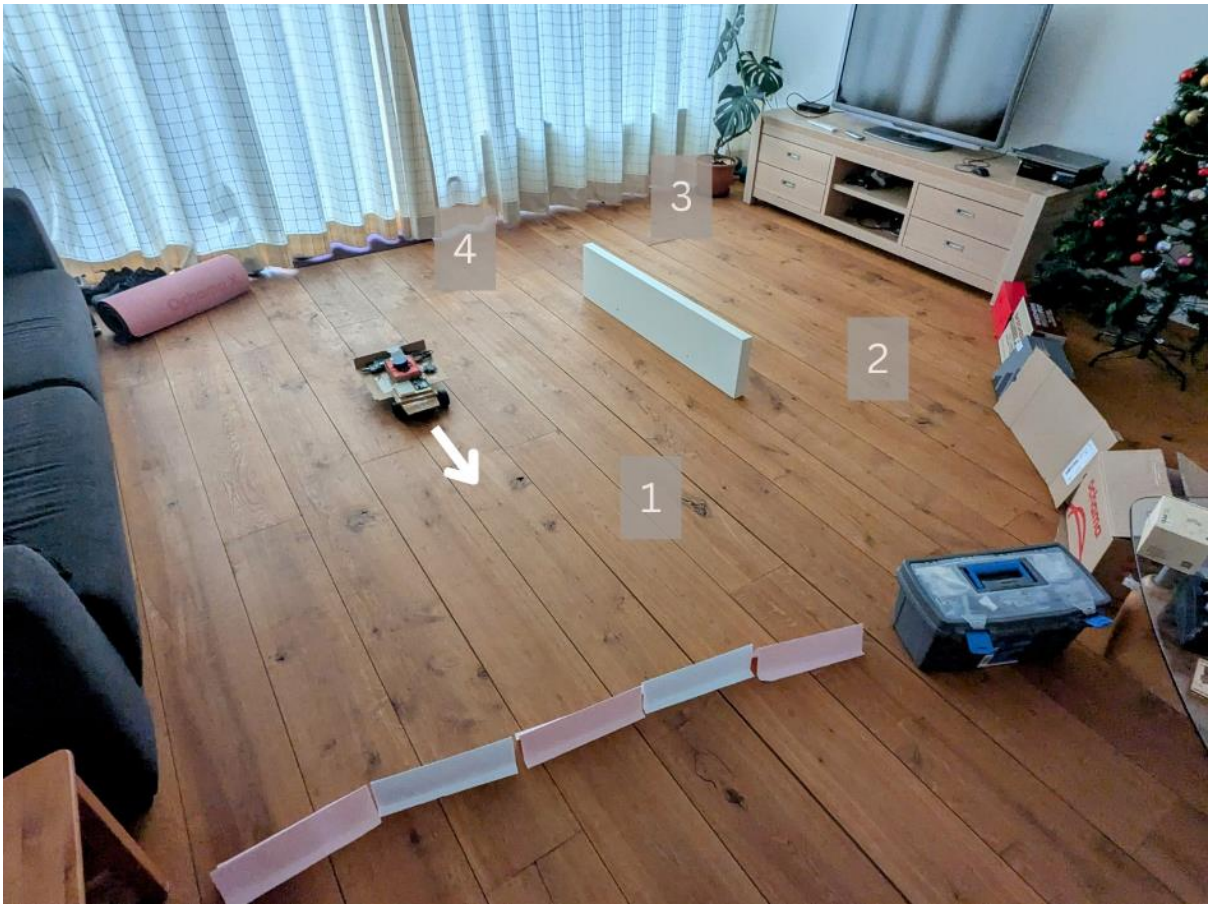
We hebben het een cirkel laten rijden op een geïmproviseerd circuit. In figuur 18 zien we de stuur en gas telemetrie van een succesvolle rit door het circuit. In figuur 19 zie je de racebaan.

We zien dat:

- Voor bochten 1, 2 en 4 de auto heel snel naar rechts stuurt en dan vervolgens naar links stuurt
- Er wordt niet meer dan 30% gas gegeven
- Er wordt midden in de bochten 1 en 3 veel gecorrigeerd door van links naar rechts te gaan
- Tijdens bocht 2 en 4 wordt gas afgenomen en aan het einde van de bocht neemt gas weer toe



Figuur 18. Gas en stuur telemetrie van de zelfrijdende auto



Figuur 19. Racebaan

H8 Discussie

8.1 Discussie simulatie

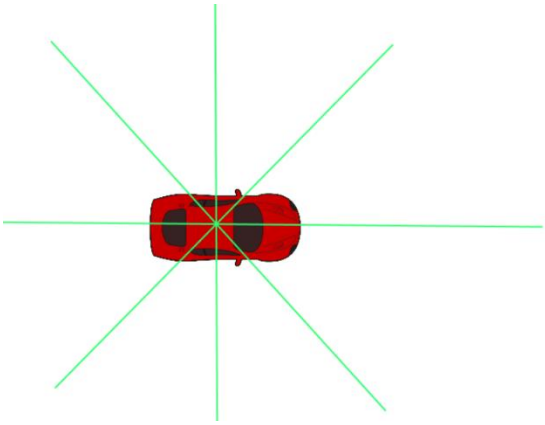
We zijn blij met hoe Daniel v3 in de simulatie rijdt. Het kan op onbekende wegen rijden met verschillende breedte en bochten. Hierdoor is bewezen dat het model echt heeft geleerd om te autorijden. We hebben gezien in figuur 16 dat de auto voor elke bocht zijn gas vermindert, dit is een teken dat Daniel v3 anticipeert op een bocht voordat hij er überhaupt is. Ook zien we dat als de auto het meeste aan het draaien is, het gas losgelaten wordt. Dit helpt de auto om scherpere bochten te maken en dat heeft Daniel v3 ook geleerd. Soms zijn er wel een beetje oscillaties in stuurbeweging in een bocht, maar de auto blijft wel recht rijden omdat de oscillaties zo snel zijn, dat het in de simulatie geen visueel verschil maakt.

De simulatie die wij hebben geprogrammeerd dient als een goede basis om latere iteraties van Daniel v3 te trainen omdat we makkelijk de stuursnelheid, snelheid en versnelling van de auto kunnen veranderen in de simulatie.

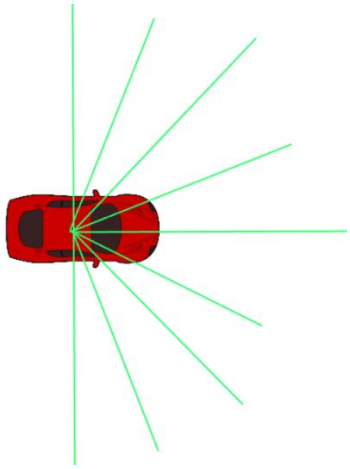
8.2 Discussie zelfrijdende auto

De auto rijdt in het echt niet zo goed vergeleken met de simulatie. Het probleem is dat de auto heel vaak heeft gecrasht voordat het een heel rondje heeft gereden. Hiervoor moesten we veel aanpassingen maken aan het AI-model voor het resultaat in 7.2 en was er geen tijd meer om de auto te testen op complexere banen. De crashes kwamen door een aantal redenen, het is heel lastig om de precieze kenmerken van de RC-auto te weten, zoals snelheid, stuur snelheid en versnelling. Waardoor de auto vaak moet corrigeren midden in een bocht zoals we zagen in figuur 18.

Ook is er het probleem dat de LiDAR sensor niet elke keer exact dezelfde hoek tussen elke afstandspunt kan scannen. Het is erg moeilijk om het model hierop te laten anticiperen. We hebben wel plannen om dit probleem op te lossen door alleen afstandspunten van 180 graden voor de auto te gebruiken (Figuur 20). Hierdoor kunnen we de 16 input neuronen van het neuraal netwerk beter benutten en compenseren we voor de meetfouten die de LiDAR af en toe maakt. Het model dat hierop wordt getraind zal Daniel v4 heten.



Figuur 20. LiDAR input voor Daniel v3



Figuur 21. Voorgestelde LiDAR input voor Daniel v4

Ook konden we heel veel tijd besparen door een Arduino car kit te kopen, deze auto's zijn gemaakt om met een computer te bestuurd worden. Dit zou ons veel tijd hebben gekost op het onderzoeken en het verbinden van de servomotoren aan de Arduino.

Voor volgende iteraties van de zelfrijdende auto kunnen we de Raspberry Pi direct aansluiten aan de servomotoren, de Raspberry Pi heeft ook GPIO-poorten die de juiste signalen en beaarding kunnen leveren om de auto te besturen. Dit zal het aantal componenten verminderen en dus complexiteit verminderen.

Om de auto sneller te laten rijden kunnen we een spoiler toevoegen voor extra grip. Met een spoiler kan een auto sneller door een bocht heen.

Voordat de PWS-markt is begonnen willen we de auto meer gaan testen op ingewikkeldere banen.

H9 – Conclusie

In dit profielwerkstuk hebben we onderzocht hoe een zelfrijdende auto kan worden gemaakt. Om deze vraag te beantwoorden, hebben we een simulatieomgeving gebouwd, een AI-model getraind met behulp van het NEAT-algoritme en een fysieke RC-auto uitgerust met een LiDAR-sensor, Raspberry Pi en Arduino. Uiteindelijk is het ons ook gelukt om een zelfrijdende auto maken dat zelf een rondje kan rijden.

Een van de belangrijkste resultaten was dat ons AI-model, Daniel v3, succesvol leerde autorijden in de simulatie. Het model anticipeerde op bochten door gas los te laten en kon onbekende banen navigeren. In de praktijk hadden we wel wat problemen, zoals afwijkingen in de LiDAR-data en de moeilijkheid om de fysieke eigenschappen van de auto nauwkeurig te simuleren. Dit resulteerde in correcties tijdens bochten en een lagere rijprestaties in vergelijking met de simulatie.

We hebben veel geleerd over kunstmatige intelligentie, hardware-integratie en de complexiteit van zelfrijdende voertuigen. Tegelijkertijd realiseerden we ons dat meer testtijd en een simpelere auto, zoals een Arduino car kit, de resultaten verder hadden kunnen verbeteren.

Ons onderzoek toont aan dat het mogelijk is om een zelfrijdende auto te bouwen die zowel in simulatie als in de echte wereld functioneert, de prestatie van de zelfrijdende auto kan verbeterd worden door realistischer te simuleren. In de toekomst hopen we dat we met Daniel v4 de huidige problemen met de zelfrijdende auto kunnen oplossen.

H10 – Reflectie

Het was voor ons beide erg leuk om aan een grote project te samenwerken waarbij we heel veel mochten programmeren. Voor ons beide was dit de meest ambitieuze programmeer project tot nu toe en we hebben erg veel geleerd.

Tijdens het programmeren van de simulatie hadden we heel veel tijd gestopt in het oplossen van een bug. De bug was dat NEAT de modellen niet goed kon laten evolueren en dus de gemiddelde fitness van de populatie stagneerde. Hierdoor konden we dus ook niet verder. De reden van de bug was dat er bij het simuleren van de LiDAR een scanfrequentie was toegevoegd om de LiDAR realistischer te simuleren. Dit beïnvloede hoe de auto stuurde in de simulatie waardoor het bijna onmogelijk was om de auto te besturen. Achteraf bleek dat de scanfrequentie simuleren helemaal geen invloed had op de prestatie van onze AI-modellen. We probeerde 2 maanden deze bug te fixen.

Het implementeren van de hardware zoals LiDAR, Arduino, Raspberry pi ging verrassend goed, wij hadden hiervoor nooit gewerkt met dit soort hardware.

Wij gaan allebei computer science studeren in de toekomst, bij groepsprojecten waar we moeten programmeren zullen we waarschijnlijk weer GitHub gebruiken om code bij te houden en delen.

Ten slotte zijn we heel trots op onszelf dat we zo een groot project hebben gekozen en er een redelijk goede resultaat eruit is gekomen dankzij onze inzet en technische vaardigheden.

Logboek

persoon	activiteit	tijdsbesteding	datum
lucas	onderzoek doen	5	08/09/2024
shengen	onderzoek doen	3	11/09/2024
lucas	voorbereiding software	8	14/09/2024
lucas	simulatie programeren	2	19/09/2024
shengen	simulatie programmeren	4	21/09/2024
lucas	simulatie programmeren	5	21/09/2024
shengen	simulatie programmeren	2	25/09/2024
shengen	simulatie programmeren	8	04/10/2024
lucas & shengen	onderzoek doen	10	15/10/2024
lucas	auto in elkaar zetten	5	25/10/2024
lucas	auto in elkaar zetten	6	02/11/2024
shengen	testen onderdelen	3	05/12/2024
shengen	testen onderdelen	3	06/12/2024
lucas & shengen	testen onderdelen	12	07/12/2024
lucas & shengen	auto uittesten	8	08/12/2024
shengen	auto fixen	8	09/12/2024
shengen	auto fixen	6	15/12/2024
lucas & shengen	auto uittesten	8	23/12/2024
lucas	auto fixen	8	28/12/2024
shengen	auto fixen	8	02/01/2025
lucas & shengen	auto uittesten	6	03/01/2025
lucas & shengen	verslag schrijven	15	03/01/2025
shengen	verslag schrijven	8	04/01/2025
shengen	verslag schrijven	8	05/01/2025
lucas	verslag schrijven	7	05/01/2025
shengen	verslag schrijven	5	14/01/2025
lucas	verslag schrijven	4	14/01/2025
lucas & shengen	verslag schrijven	10	15/01/2025
lucas & shengen	verslag schrijven	15	16/01/2025
		Totaal: 200 uur	

De uren waarbij we hebben samengewerkt zijn gecombineerd.

Bronvermelding

Arduino. (2024). *Servo documentation.*
https://docs.arduino.cc/libraries/servo/#Usage/Examples

Arthur Arnx. (2019). *First neural network for beginners explained (with code).*

https://towardsdatascience.com/first-neural-network-for-beginners-explained-with-code-4cfd37e06eaf

Himoto. (z.d.) *Transmitter MT-303TX operating instructions*.

https://distributions.com.ua/files/%D0%98%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%86%D0%B8%D1%8F_%D0%BA_%D0%B0%D0%BF%D0%BF%D0%B0%D1%80%D0%B0%D1%82%D1%83%D1%80%D0%B5_Himoto_MT-303_%D0%B4%D0%BB%D1%8F_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D0%B5%D0%B9_1_18_%D0%B1%D0%BA__1_10__1_8_%28%D1%81%D0%BA%D0%B0%D0%BD._%D0%B0%D0%BD%D0%B3%D0%BB.%29_21.01.21.pdf

Kenneth O Stanley, Risto Miikkulainen. (2002). *Evolving neural networks through augmenting topologies.*

https://nn.cs.utexas.edu/downloads/papers/stanley.cec02.pdf

Raspberry Pi. (2024). *GPIO.*
https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#gpio

Robert MacWha. (2021). *Evolving AIs using a NEAT algorithm.*

https://macwha.medium.com/evolving-ais-using-a-neat-algorithm-2d154c623828

Roboticsbackend. (z.d.). *Raspberry Pi Arduino Serial Communication – Everything You Need To Know.*

https://roboticsbackend.com/raspberry-pi-arduino-serial-communication/

Senetra. (z.d.). *What is a common ground.*

https://www.sentera.eu/en/faq/g/what-is-a-common-ground/993#:~:text=Having%20a%20common%20ground%20ensures,for%20proper%20operation%20and%20safety.

Trevor Burton-McCreadie. (2024). *The NEAT Algorithm: Evolving Neural Networks.*

https://blog.lunatech.com/posts/2024-02-29-the-neat-algorithm-evolving-neural-network-topologies

Wikipedia. (2024). *Ackermann steering geometry.*

https://en.wikipedia.org/wiki/Ackermann_steering_geometry

Wikipedia. (2024). *Lidar.*

https://en.wikipedia.org/wiki/Lidar