Projektdokumentation für das

Self Driving Car

Entwickelt von Tim Bader

Sonntag, 12. Januar 2020

Inhaltsverzeichnis

Projektbeschreibung (Kurzfassung)

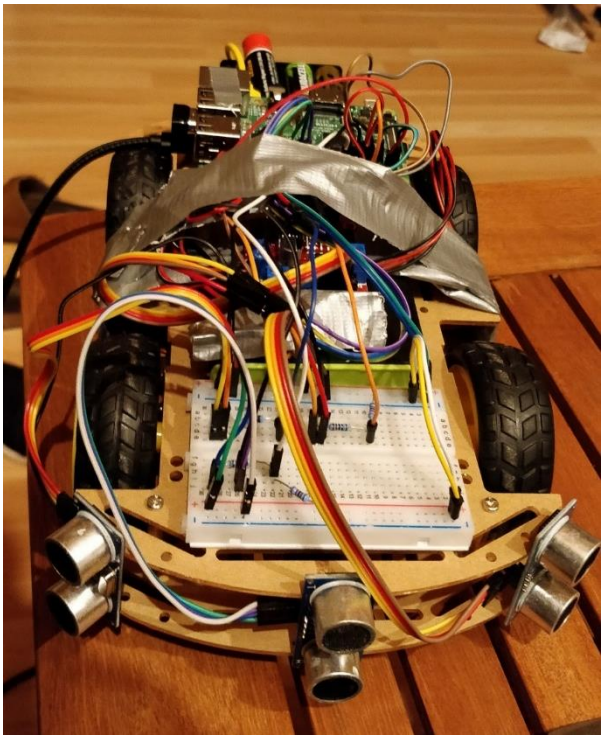
Das „Self Driving Car“ Projekt beschreibt den Versuch, eine künstliche Intelligenz in ein Spielzeugauto einzupflanzen. Das selbstgebaute RC Car ist mit einem Bordcomputer, genauer gesagt einen Raspberry PI 3B, und drei Ultraschallsensoren ausgestattet.

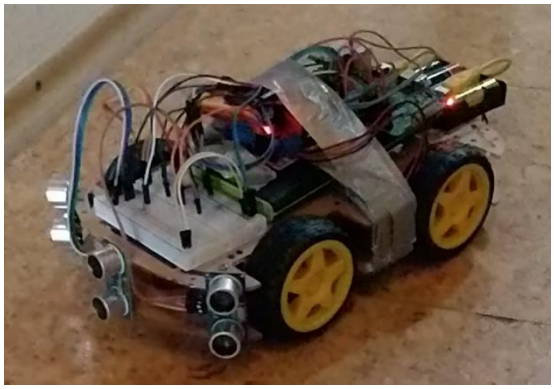
Damit das Auto selbstständig fahren kann, muss der Raspberry PI einen speziellen Algorithmus ausführen. Aber das ist nicht nur ein einfacher Algorithmus, sondern eine vortrainierte künstliche Intelligenz. Das „Gehirn“ dieser KI besteht aus vielerlei Komponenten. Wichtig für dieses Beispiel sind allerdings die „Weights“, die nichts anderes als ein Haufen von statischen Zahlen sind. Denn Weights managen mithilfe von mathematischen Operationen die Verknüpfungen des Gehirns.

Diesem Gehirn werden nun die Werte der Ultraschallsensoren übergeben, worauf hin der Raspberry PI die Motorgeschwindigkeit und den Rotationswert berechnen kann. Das alles passiert mehrmals pro Sekunde.

In zuvor ausgeführten Simulationen wird das Spielzeugauto nachgebaut und mit demselben neuronalen Netz ausgestattet. Auch hier soll das Auto eine möglichst lange Strecke fehlerfrei bewältigen. Der einzige Unterschied ist, dass die Weights zu Beginn zufällig generiert wurden. Nach jeder Generation eines simulierten Autos werden die Weights angepasst. Nur die Weights der besten zwei Autos werden miteinander vermischt.

Lässt man diese Computersimulation nun mehrere Stunden lang arbeiten, erhält man früher oder später eine optimale Version des Autos. Um das trainierte neuronale Netz in der Realität anzuwenden, muss man die Weights des besten Autos übernehmen. Dafür wird das Gehirn auf dem Raspberry PI nachgebaut und mit den optimierten Weights befüllt und voila – ein selbstfahrendes Auto.

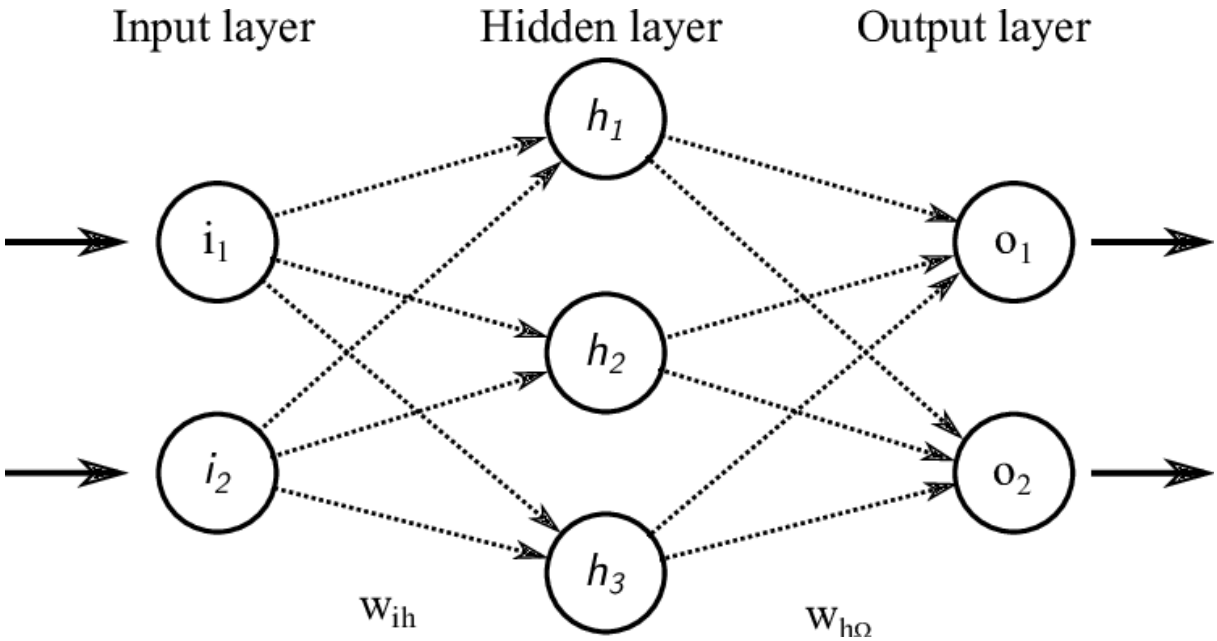




Wie funktioniert ein neuronales Netzwerk?

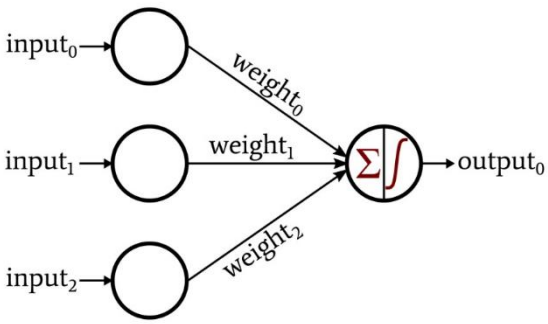
Im Jahre 2012 fand ein weltweiter ‚Image Recognition Algorithm'-Wettbewerb statt. Dort gelang es einem kanadischen Team der University of Toronto, mithilfe des ersten „deep convolutional neural network", eine Fehlerrate von nur 16% zu erzielen. Seit diesem Zeitpunkt hat die künstliche Intelligenz immer mehr an Bedeutung gewonnen.

Ein Neuronales Netzwerk ist die Grundlage für sämtliche Anwendungsbereiche der KI. Es besteht aus Neuronen, Weights und vielerlei mathematischer Funktionen.



Die Elemente des Netzwerks sind in verschiedenen Layern angeordnet. Dem Input Layer werden Werte von z.B. Messungen übergeben, welches diese dann an die Hidden Layers weiterleitet. Irgendwann gelangen diese Werte über die Hidden Layers dann an das Output Layer, welches die bearbeiteten Werte zurückliefert.

Die Neuronen (Kreise im Bild) dienen als Zwischenspeicher für jeweils genau einen Zahlenwert. Neuronen sind mithilfe von Weights mit jedem Neuron im folgenden Layer verbunden. Der Zahlenwert eines Neurons ist von drei Faktoren abhängig: den Werten der vorherigen Neuronen, den Weights und einer Aktivierungsfunktion.



Um den Wert eines Neurons zu berechnen, muss man sich zunächst das vorherige Layer anschauen.
Jedes Neuron des vorherigen Layers wird mit dem dazugehörigen Weight multipliziert. Diese Ergebnisse fasst man in einer Summe zusammen und übergibt diese einer Aktivierungsfunktion.

Die Aktivierungsfunktion und die Weights sind statisch und werden während eines Durchlaufs nicht verändert. Weights werden nur beim Training durch sogenannte Backpropagation verändert. Die Aktivierungsfunktionen müssen beim Erstellen des neuronalen Netzes manuell des Anwendungsbereichs entsprechend angepasst werden. Häufige Aktivierungsfunktionen sind ReLu [0; ∞] und Sigmoid [-1; 1].

Bei fehlenden Vorkenntnissen ist es empfehlenswert, sich die ersten Videos dieser Videoplaylist anzusehen:
https://www.youtube.com/watch?v=gZmobeGL0Yg&list=PLZbbT5o_s2xq7LwI2y8_QtvuXZedL6tQU

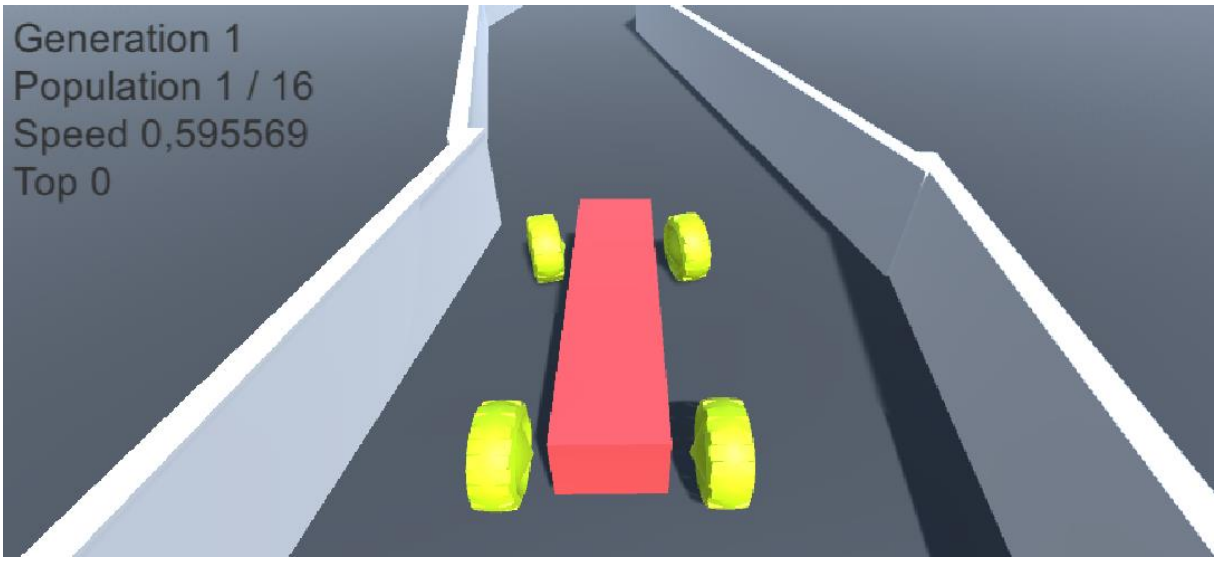
Umsetzung

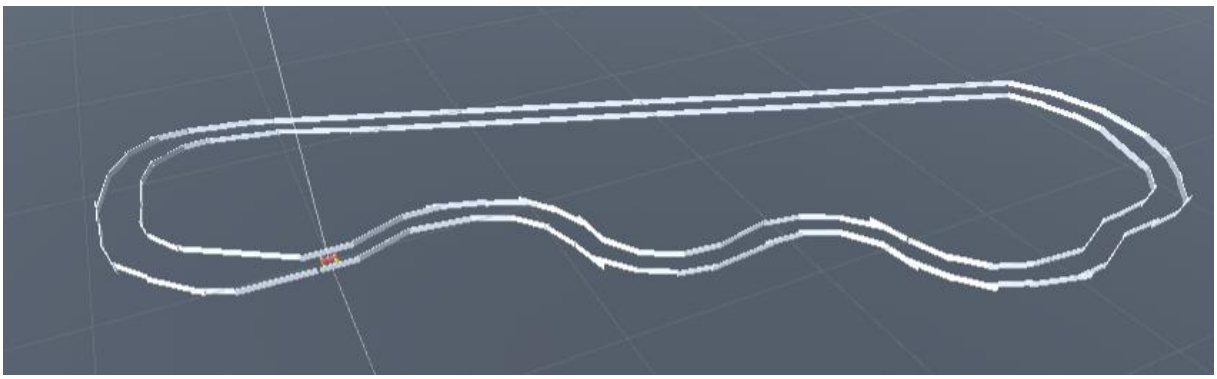
Simulation in Unity 3D

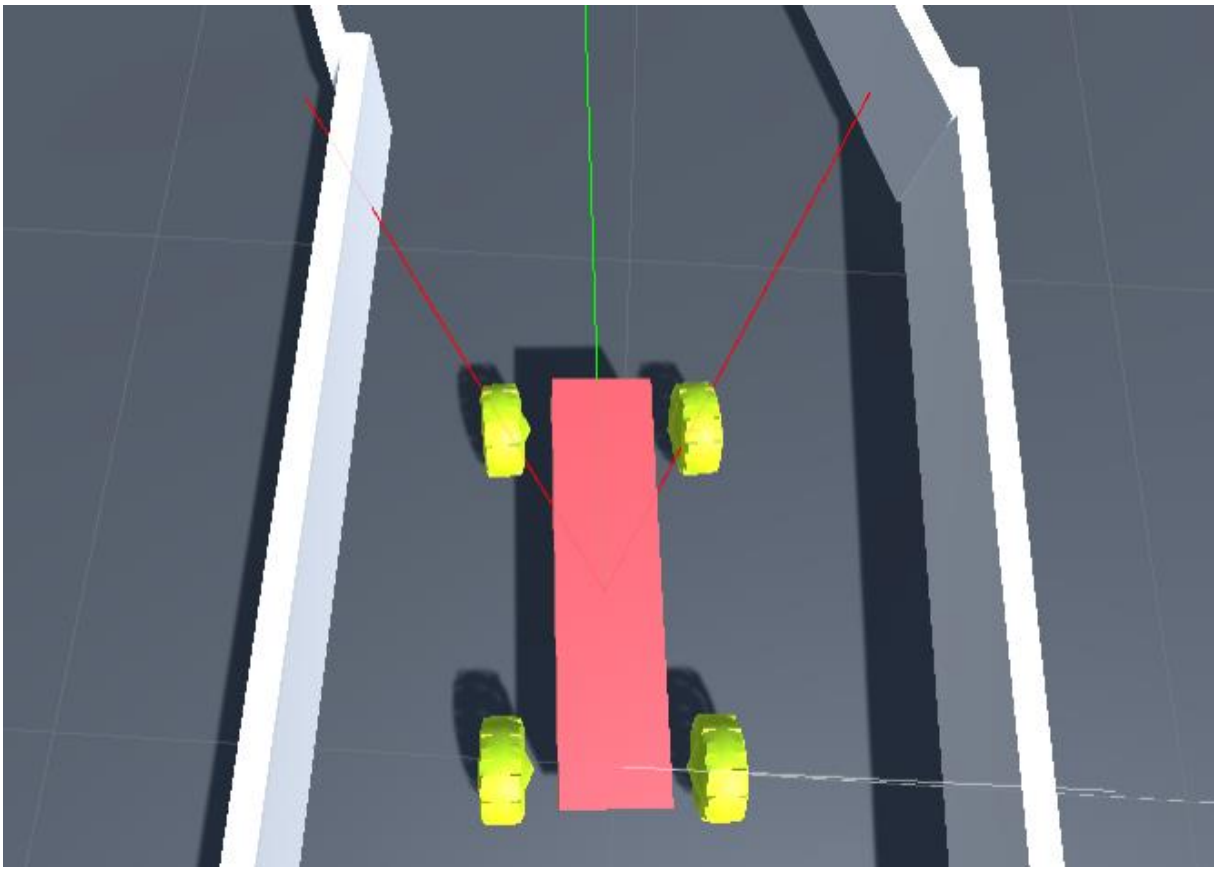
Es gibt mehrere Arten, eine künstliche Intelligenz zu trainieren. Die wohl einfachste Methode ist das „trial-and-error“-Verfahren, das in Bezug auf Deep Learning auch als „Reinforcement Learning“ bezeichnet wird. Da Simulationen immer dieselben Umstände bieten und leicht anpassbar sind, sind diese demzufolge das perfekte Werkzeug, um ein Netzwerk solcher Art möglichst schnell zu trainieren.

Für das Self Driving Car Projekt habe ich auf der Simulation von dem YouTuber Mateusz Bączek aufgebaut. Er hat Dinge, wie z.B. ein Auto, eine Teststrecke und die Evolutionslogik für das Verbessern der KI bereits in einer Unity 3D Projektdatei umgesetzt.

Meine Aufgabe war es, die Simulation so anzupassen, dass ich das fertige neuronale Netzwerk extrahieren und auf das reale RC Auto anwenden kann.







Die Simulation ist mit sehr viel C# Code hinterlegt. Es würde den Rahmen sprengen, diesen hier zu erläutern. Jedoch gibt es im Code selbst viele Bezeichnungen.

Das Grundprinzip der Simulation orientiert sich am Darwinismus: der stärkere überlebt.
Die simulierten Autos sind in Generationen unterteilt, die jeweils eine Größe von 16 haben.
Immer, wenn ein Auto gegen eine Wand fährt oder stehen bleibt, wird die Simulation zurückgesetzt und das nächste Auto wird generiert.
Wiederholt man diesen Prozess 16 Mal, so hat man eine ganze Generation durchlaufen. Am Ende jeder Generation werden die Ergebnisse ausgewertet. Hierbei wird Strecke Pro Zeit miteinander verrechnet. Nur die zwei Autos, die am besten abschneiden, werden für die nächste Generation miteinander gekreuzt.
Das Kreuzen zweier Autos läuft wie bei Organismen in der Natur ab. Die DNA der Eltern, in diesem Fall die Weights, werden jeweils zufällig miteinander kombiniert.
Schneidet eine Generation schlechter ab wie die vorherige, so wird diese ausgeblendet und die Weights der alten Eltern werden erneut verwendet. Die erste Generation wird zufällig generiert.
Nach jeder Generation werden die Weights des besten Autos in eine Textdatei kopiert. Diese lassen sich dann beliebig weiterverwenden.
Diesen Gegebenheiten zufolge muss jedes der Autos mit einem eigenen neuronalen Netzwerk ausgestattet sein. Die Größe des Netzwerks hält sich mit seinen 3 Layern und insgesamt nur 11 Neuronen in Grenzen.

Python 3 Code für den Raspberry PI

Der Bordcomputer des Prototyps ist ein Raspberry PI 3 B, der nichts anderes als ein Microcomputer ist. Das Betriebssystem basiert auf Linux Ubuntu und die verwendete Skriptsprache ist Python 3.7.

Das Ziel war es, das neuronale Netz die Hardware zu steuern zu lassen. Wenige manuelle Befehle wie „Start“, „Pause“, „Resume“ und „Stop“ sollen über die Tastatur manuell ausführbar sein.

Damit die Tastaturinputs registriert werden können, müssen diese in einem Loop dauerhaft abgefragt werden. Da die Sensorabfragen und Motorkommandos ebenfalls getaktet ablaufen müssen, hat es sich angeboten, diese in demselben Loop unter Berücksichtigung der manuellen Befehle miteinzubinden. Das Resultat ist ein großer Mainloop.

Um eine gewisse Übersicht zu wahren, wird im Code hauptsächlich mit Methoden gearbeitet:

-**getDistances()**: misst die Abstände und gibt ein Array mit drei Kommazahlen zurück.

-**network(array[0,0,0])**: wertet die Abstände in einem neuronalen Netzwerk aus und gibt anschließend die zwei Zahlenwerte für Steering und Speed zurück.

-**move(l, r)**: l und r sind die Motorgeschwindigkeiten für die jeweilige Seite des Prototyps. Die Methode wandelt die Werte um und steuert die vier Motoren wie einen Panzer.

```
while True:
    results = network(getDistances())
    steering = max_steering_angle*(results[0]-0.5)*2
    speed = max_motor_torque*results[1]+0.2

    #combining speed and steering variables to one with math
    #because real wheels are static and cannot rotate
    l = speed
    r = speed
    if steering > 0.1: #right
        r = r - steering/100
    if steering < -0.1: #left
        l = l + steering/100
    move(l, r)

    #PAUSE
    if keyboard.is_pressed("p"):
        print("Paused. Press R to resume.")
        keyboard.wait("r")
    #EXIT
    if keyboard.is_pressed("x"):
        print("Exiting...")
        break
    time.sleep(loop_time)
```

Die steering und speed Variablen müssen angepasst werden, um dieselben Bedingungen wie in der Simulation zu schaffen.

Da das Fahrzeug kein richtiges Lenksystem besitzt, muss die Rotation über die Motorgeschwindigkeit der jeweiligen Seiten geregelt werden. Hierzu werden Steering und Speed abhängig voneinander in die neuen Variablen l und r umgewandelt.

Mit den Tasten „p“ und „r“ kann man die Ausführung des Codes pausieren und wieder fortsetzen. „x“ beendet das Programm.

Die Methoden haben neben dem Standardcode auch einige nennenswerte Stellen.

Eine davon findet sich z.B. bei der Abstandsmessung innerhalb von „**getDistances()**“.

```
pulse_start1 = time.time()
pulse_end1 = time.time()
GPIO.output(TRIG1, False)
time.sleep(0.06)
GPIO.output(TRIG1, True)
time.sleep(0.00001)
GPIO.output(TRIG1, False)

while GPIO.input(ECHO1)==0:
    pulse_start1 = time.time()
while GPIO.input(ECHO1)==1:
    pulse_end1 = time.time()

pulse_duration1 = pulse_end1 - pulse_start1
distance1 = pulse_duration1 * 17150
distance1 = round(distance1, 2)
```

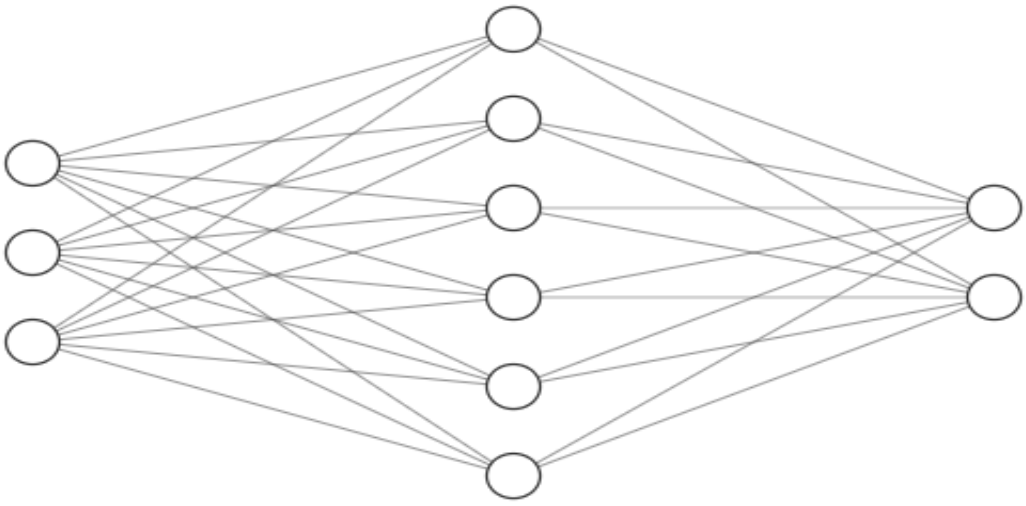
Der Ultraschallsensor wird durch ein kurzes Stromsignal ausgelöst. Sobald er das Echo seiner eigenen Welle empfängt, sendet er ein kurzes Stromsignal zurück.
Um die Distanz zu erhalten, muss man zunächst die Differenz zwischen den Zeitpunkten der Stromsignale berechnen. Dadurch erhält man die Reisezeit der Ultraschallwelle.
Eine Ultraschallwelle ist so schnell wie die Schallgeschwindigkeit, also 34300cm/s. Da die vorhin gemessene Zeit der Reisezeit zum Hindernis hin und zurück entspricht, muss man die Schallgeschwindigkeit halbieren. Erst jetzt kann man diese mit unserer gemessenen Zeit multiplizieren, um dann die Distanz in Zentimeter zu erhalten.

Die wohl interessanteste Stelle im Code ist aber die Implementation des neuronalen Netzwerks.

Drei Abstandswerte werden der **network()** Methode übergeben. Allein damit sollen jetzt Geschwindigkeit und Rotation berechnet werden.

Zur Erinnerung: die Weights wurden von dem Auto, das in der Simulation am besten abgeschnitten hat, übernommen.

Zunächst sollte man sich aber das Diagramm des neuronalen Netzwerks erneut vor Augen halten. Die Input-Werte werden von Layer zu Layer weitergegeben und neu berechnet. Dieses Mal sind alle Werte von Beginn an statisch, man könnte den Code also auch als große Gleichung formulieren.



Input Layer $\in \mathbb{R}^3$ Hidden Layer $\in \mathbb{R}^6$ Output Layer $\in \mathbb{R}^2$

```
def sigmoid(x): #sigmoid function which returns a value between -1 and 1
    return 1/(1 + math.exp(-float(x)))

def network(inputs): #inputs run through a pre-trained neural network
    parameters = [3, 6, 2] #nn architecture
    #best generation of weights from pre-trained neural network
    weights = [[[-0.980175495147705, -0.186096787452698, 0.80379843711853, 0.948149442672729, -0.180595278739929, -0.904619455337524],
                [-0.30886709690094, -0.582291841506958, -0.6927809715271, 0.557246446609497, 0.611737489700317, -0.190137267112732],
                [-0.756172895431519, 0.545915603637695, -0.368541121482849, -0.635722160339355, 0.956294536590576, 0.745823383331299]],

               [[-0.972720861434937, 0.146159529685974],
                [0.563815593719482, 0.683503150939941],
                [0.439907282590866, 0.566266775131226],
                [0.725433349609375, -0.598716497421265],
                [-0.830876111984253, -0.262409806251526],
                [-0.68674635887146, -0.45588481426239]]]
    for i in range(2): #layers
        outputs = [0.0] * parameters[i+1]
        for j in range(len(inputs)): #neurons
            for k in range(len(outputs)): #weight of each neuron
                outputs[k] += inputs[j] * weights[i][j][k] #first part of math is multiplication of input and weight
        inputs = [0.0] * len(outputs)
        for l in range(len(outputs)):
            inputs[l] = sigmoid(outputs[l] * 5) #second part of math is sigmoid function, output becomes input for next loop
    return inputs
```

Mit den for-Loops werden die verschiedenen Layer und deren Neuronen in logischer Reihenfolge durchlaufen und verarbeitet. Im tiefsten Loop werden die Output-Werte des aktuellen Layers als Input-Werte für das darauf folgende Layer festgelegt. Genau an dieser Stelle werden die Werte mit den aktuellen Weights multipliziert und anschließend durch die Sigmoid Funktion proportional auf eine Größe zwischen 0 und 1 reduziert – genau wie in der Simulation.

Am Ende bleiben 2 Werte übrig. Das sind dann Geschwindigkeit und Rotation, mit denen andere Methoden weiterarbeiten können.

Benötigte Hardware und deren Einrichtung

Der Hardware-Teil erforderte einiges an Übung und Kenntnissen in der Elektrotechnik.

Auf dem Raspberry PI wird für die Steuerung der GPIO-Outputs die RPi Library verwendet.

Die Stromversorgung erfolgt über zwei verschiedene Energiequellen: 4 AA Haushaltsbatterien für die Motoren und eine 5V Powerbank für den Raspberry PI.

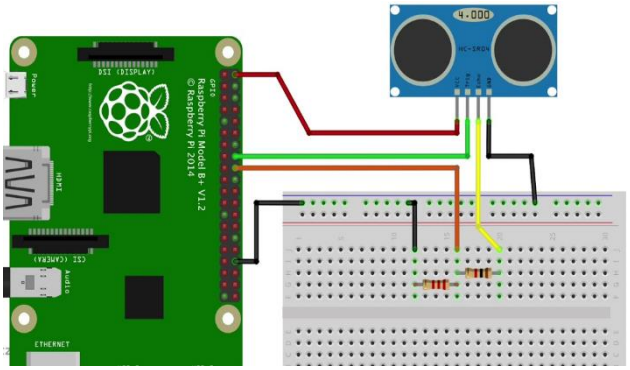
Die Inkonsistenz der Powerbank löst des Öfteren Fehlfunktionen am Raspberry PI aus und sollte langfristig mit einer hochwertigeren Lösung ausgetauscht werden.

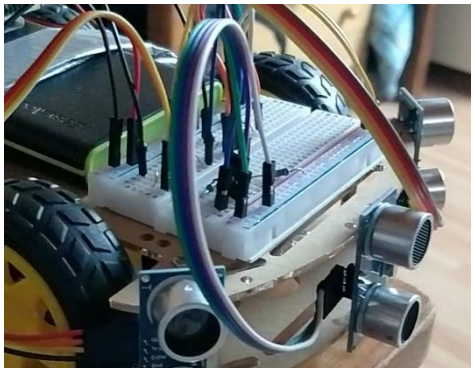
Damit die Motoren nur dann mit dem Strom der AA-Batterien versorgt werden, wenn sie sich auch wirklich bewegen sollen, wird die Stromzufuhr mit einer L298N Dual H-Brücke kontrolliert. Diese ist zugleich auch mit dem Raspberry PI über 4 verschiedene Kabel verbunden. Mit diesen 4 Kabeln lassen sich je nach Stromfluss die Stromzufuhr für die 4 Motoren steuern und ermöglichen somit Aktionen wie geradeaus fahren, auf der Stelle drehen usw.

Um die Distanzen zu messen, wurden 3 Ultraschallsensoren des Typs HC-SR04 verwendet. Jeder dieser Sensoren besitzt 4 Anschlussstellen: Gnd (Ground), Echo (aktiv beim Empfangen des Echos), Trig (wenn aktiv dann löst Sensor Ultraschallwelle) und VCC (Stromversorgung).

Hierbei gilt zu beachten, dass der Stromstoß des Echo-Outputs eine 5V Spannung beträgt. Die GPIO Inputs des Raspberry PI sind jedoch nur auf 3V ausgelegt, weshalb mithilfe von zwei verschiedenen Wiederständen die Spannung reduziert werden muss.

Die drei Ultraschallsensoren wurden alle mit demselben Prinzip wie in diesem Schaubild angeschlossen.





Herausforderungen

Die Entwicklung des Self Driving Cars brachte viele Herausforderungen mit sich.

Eine der größten war mit Sicherheit der Abgleich zwischen dem simulierten Auto und dem realen Auto.
Das Auto in der Simulation kann über die Vorderachse steuern, während der Prototyp sich wegen Materialkosten nur wie ein Panzer drehen kann. Zudem musste man ein gesundes Verhältnis zwischen Geschwindigkeit, Gewicht und Reibungskräften herstellen. Da dies praktisch nur bedingt möglich war, konnte der andere Teil durch eine besondere Einbettung in den Python Code berichtigt werden.
Ein konkretes Beispiel dafür ist z.B. die Geschwindigkeitsregelung. In der Simulation wird die Schnelligkeit des Autos mit einem Wert zwischen 0 und 1 festgelegt. Die realen Motoren können nicht gedrosselt werden, für sie gibt es nur ein an oder aus. Dementsprechend war es nur über eine im Code implementierte Zeittaktung möglich, sowohl Geschwindigkeit als auch Richtung gleichzeitig zu beachten. Darunter leiden die eigentliche Fahrzeuggeschwindigkeit und auch die Genauigkeit.

Eine weitere Herausforderung boten auch die Abstandssensoren. Diese liefern nämlich nicht immer genaue Werte.
Die Minimalreichweite beträgt aus praktischer Erfahrung ca. 1,5cm, was bei einem engen Parkour zu Problemen aufgrund von falschen Messwerten führt.
Doch auch auf „normalen“ Strecken machen die Abstandssensoren öfters mal Probleme. Treffen die Ultraschallwellen z.B. im 30° Winkel auf eine Wand, so wird die Welle größten Teils von der Wand in eine andere Richtung weiter reflektiert. Dieser Teil der Hauptwelle gelangt über Umwege wieder zurück zum Sensor, was wegen der längeren Reisezeit eine falsche und größere Distanz zurückliefert.

Potenzielle Erweiterungen

Das Self Driving Car ist alles andere als ausgereift. An vielen Stellen muss noch gefeilt werden, an anderen bieten neue Möglichkeiten an.

Höhere Genauigkeit durch Softwareverbesserungen

Das Self Driving Car hat noch viel Luft nach oben.
Selbst ohne Hardwareupgrades ließe sich eine höhere Genauigkeit der Autonomität erzielen.
Für die Verbesserung der Software gibt es zwei konkrete Ansatzpunkte. Der erste wäre die Verbesserung der Simulation.
Hier lässt sich einiges verbessern. Das offensichtlichste ist wohl die Trainingsstrecke, die nicht wirklich abwechslungsreich ist. Man könnte das Auto auf mehreren Strecken oder gar auf einer zufällig generierten Strecke ohne Runden trainieren. Dadurch wird sogenanntes „overfitting“ verhindert, was bei dem jetzigen Prototyp nur bedingt Einfluss nimmt. Auf das muss man aber genauer achten, wenn man Veränderungen an dem neuronalen Netzwerk vornimmt. Denn genau das wäre eine weitere Verbesserung der Simulation. Das Netzwerk besteht aus nur einem Hidden Layer und sehr wenigen Neuronen. Fügt man zusätzliche Neuronen und Layer hinzu, so werden die Output Daten nach entsprechendem Training um einiges genauer.
Der zweite Ansatzpunkt ist der Python Code des Bordcomputers. Damit die Output-Werte des neuronalen Netzwerks umgewandelt werden können, müssen einige statische wenn-dann Verzweigungen eingebaut werden. Diese sind simpel und übersichtlich gehalten, worunter allerdings die Genauigkeit leidet. Am besten wäre ein möglichst breites Spektrum an abgedeckten Fällen. Das ist notwendig, da die Motoren nicht im Einklang mit dem neuronalen Netzwerk sind und somit eine Schnittstelle mit möglichst wenig Informationsverlust gegeben ist.
Ein konkretes Beispiel dafür wäre folgendes:
L zeigt 0.65 an, während R 0.45 anzeigt. Der Prototyp fährt in diesem Fall weiter gerade aus. In den meisten Fällen ist das in Ordnung, doch ab und zu hätte er wohl doch lieber nach rechts einschlagen sollen.
Durch das temporäre Speichern der vorherigen Aktionen unter Miteinbezug der aktuellen Abstandswerte ließen sich diese Spezialfälle deutlich öfters abfangen, was die Autonomität des Fahrzeugs positiv beeinflusst.

Manuelle Steuerung

Die Steuerung des Prototyps erscheint zunächst sehr kontrovers, da es nur einen autonomen und keinen manuellen Modus gibt.
Beim Hinzufügen von einer manuellen Kontrolle müsste man sich zunächst auf ein Steuerelement festlegen. Der einfachste Weg wäre die Verwendung der bereits vorhandenen Funktastatur, da dann keine WiFi oder Bluetooth Schnittstelle eingerichtet werden müsste.
Der autonome Modus wird über den Hauptloop verwaltet. Durch das Hinzufügen eines zweiten Loops könnten dort z.B. die Tastaturinputs der Pfeiltasten verwaltet werden. Da dieser Loop unabhängig wäre, könnte man die langsame Taktung der Motoren umgehen und das Fahrzeug somit schneller machen. Per Knopfdruck wäre es möglich, zwischen den Loops zu wechseln und somit den Modus je nach Belieben festzulegen.
Der Prototyp besitzt keine manuelle Steuerung, da diese für die Schaustellung von trainierter künstlicher Intelligenz als irrelevant erscheint und den Code nur unnötig unübersichtlich machen würde.

Image Recognition mit Kamera

Der Prototyp besitzt eine sehr primitive Version eines neuronalen Netzwerks. Grund dafür sind nur die 3 Abstandssensoren, die keine sehr hohe Datenlast darstellen.
Tauscht man die Ultraschallsensoren gegen eine Kamera aus, öffnet sich eine Welt voller neuer Möglichkeiten.
Hochgerechnet liefert ein Full HD Video mit 30 FPS innerhalb einer Sekunde (4,5 Mio Bytes) 500.000 Mal mehr Daten als die momentan verwendeten Sensoren zusammen (9 Bytes). Je nach nutzbarer Rechenleistung kann die Videoqualität auch herunterskaliert werden. Durch richtiges Training könnte man dadurch eine nahezu nicht existente Fehlerquote erreichen.
Zusätzlich kann man mithilfe der Kamera Features, wie z.B. das Erkennen von Stoppschildern, hinzufügen.
Das Neuronale Netz müsste dementsprechend umstrukturiert werden. An Stelle von drei Layern mit jeweils nur wenigen Neuronen bräuchte man schätzungsweise mindestens 7-10 Layer mit jeweils hunderten bis tausenden von Neuronen. Allein schon das Einlesen der Bilddaten bedeutet, dass die Anzahl der Neuronen des Input Layers der Pixelanzahl der einzelnen Frames entsprechen muss. Bei einer Videoqualität von 28x28 bräuchte man demnach also ein Input Layer mit 784 Neuronen.
Es wird deutlich, dass dieses Vorhaben gewaltige Dimensionen annimmt.
Die aktuell angewandte darwinistische Lerntechnik ist wegen dieser Komplexität alles andere als effizient. Für die Realisierung eines solchen neuronalen Netzwerks wäre es zu empfehlen, Tensorflow in Kombination mit PyTorch zu verwenden. Diese bieten das richtige Framework, neuronale Netze für Bildverarbeitung (Convolutional Neural Networks) zu entwickeln und diese auch Big Data gerecht zu trainieren.
Unity 3D bietet seit Neustem die Möglichkeit an, genau diese Frameworks mithilfe des „ml-agents-master“ Kits in die Simulation miteinzubinden. Somit ist der Grundstein für das nächste Level autonomer Projekte gelegt.
Wendet man diese Theorie in großem Stil mit zusätzlichen Kameras, Sensoren und Algorithmen an, so ist man an der Front des autonomen Fahrens angekommen. Führend in diesem Gebiet ist z.B. der amerikanische Autohersteller Tesla mit seiner Autopilot-Funktion

Schlusswort

Von Beginn ab war das Self Driving Car Projekt zu Lernzwecken gedacht. Ziel war es, in der realen Welt einen Gebrauch von in einer Simulation trainierten künstlichen Intelligenz zu machen und dies zu analysieren. Der Prototyp war nie dazu gedacht, autonomes Fahren perfekt zu beherrschen, sondern sollte als Anschauung und Basis für weitere Projekte dienen.

Rückblickend auf diese Zielsetzung war das Projekt ein voller Erfolg. Gerade in der Informatik ist es immer wieder schön zu sehen, wie sich stunden- bzw. wochenlange Arbeit am Ende ausgezahlt machen. Das besondere hierbei ist, dass man das Ergebnis direkt vor Augen hat.
Persönlich kann ich sagen, dass ich bereichsübergreifend grundlegende neue Kenntnisse sowohl in der IT, als auch in der Elektrotechnik gewonnen habe – und das ist das wichtigste für mich.

Quellen

Hardware Tutorials:

HC-SR04 Ultrasonic Rangefinder (Raspberry Pi) https://www.youtube.com/watch?v=sXJjfEisjpo (04.01.2020)

Raspberry pi with Python for Robotics 2 - Motor Control https://www.youtube.com/watch?v=pbCdNh0TiUo (04.01.2020)

Raspberry pi with Python for Robotics 3 - Connecting 4 motors https://www.youtube.com/watch?v=T4tp14zNnYI (04.01.2020)

Raspberry pi with Python for Robotics 4 - Forward and Reverse Motors https://www.youtube.com/watch?v=TtKUSFqtFmE&t (04.01.2020)

Raspberry pi with Python for Robotics 6 - Pivot RC Car https://www.youtube.com/watch?v=1fbwgBpS3ik (04.01.2020)

Simulation Preset:

Neural network driving a car in Unity https://www.youtube.com/watch?v=Jol0hnGbyN4 (04.01.2020)

Simulation Tutorial(s):

Self Driving Car Simulation Unity 3D using Genetic Algorithms and Neural Networks - Unity 3D & C# https://www.youtube.com/watch?v=m8fYPy9eiOo&list=PLVxq4HUq7dJM9HmxUCcrv3mrFjay_vVd9 (Playlist) (04.01.2020)

Bildquellen:

Neural Network full size https://www.allaboutcircuits.com/technical-articles/how-to-perform-classification-using-a-neural-network-a-simple-perceptron-example/ (07.01.2020)

Neural Network Small https://www.researchgate.net/figure/Example-for-an-artificial-neural-network-with-two-input-neurons-two-hidden-neurons-and_fig1_289479445 (07.01.2020)

Ultrasonic Sensor Diagram https://www.youtube.com/watch?v=L90WS-ptnvI (12.01.2020)

Textquellen:

Image Recognition Competition 2012 https://www.technologyreview.com/s/530561/the-revolutionary-technique-that-quietly-changed-machine-vision-forever/ (04.01.2020)