jugend forscht

Wettbewerbsjahr:	**2018**
Bundesland:	**Nordrhein-Westfalen**
Sparte:	**Schüler experimentieren**
Fachgebiet:	**Mathematik / Informatik**
Projektbetreuer:	**Dr. Sarah Behrens, Freiherr-vom-Stein-Gymnasium** **Marcel Hahn, Freiherr-vom-Stein-Gymnasium**
Erstellungsort des Projekts:	**Freiherr-vom-Stein-Gymnasium, 48161 Münster (Gymnasium)**
Patentanmeldung:	**nein**
Arbeiten mit Tieren:	**nein**
Projektnummer:	**154840**
Zul. hochgeladen:	**18.03.2018 13:23**
Name Regionalw.:	**Münsterland**

Projekttitel: **Selbstfahrende Autos mit neuronalen Netzen**

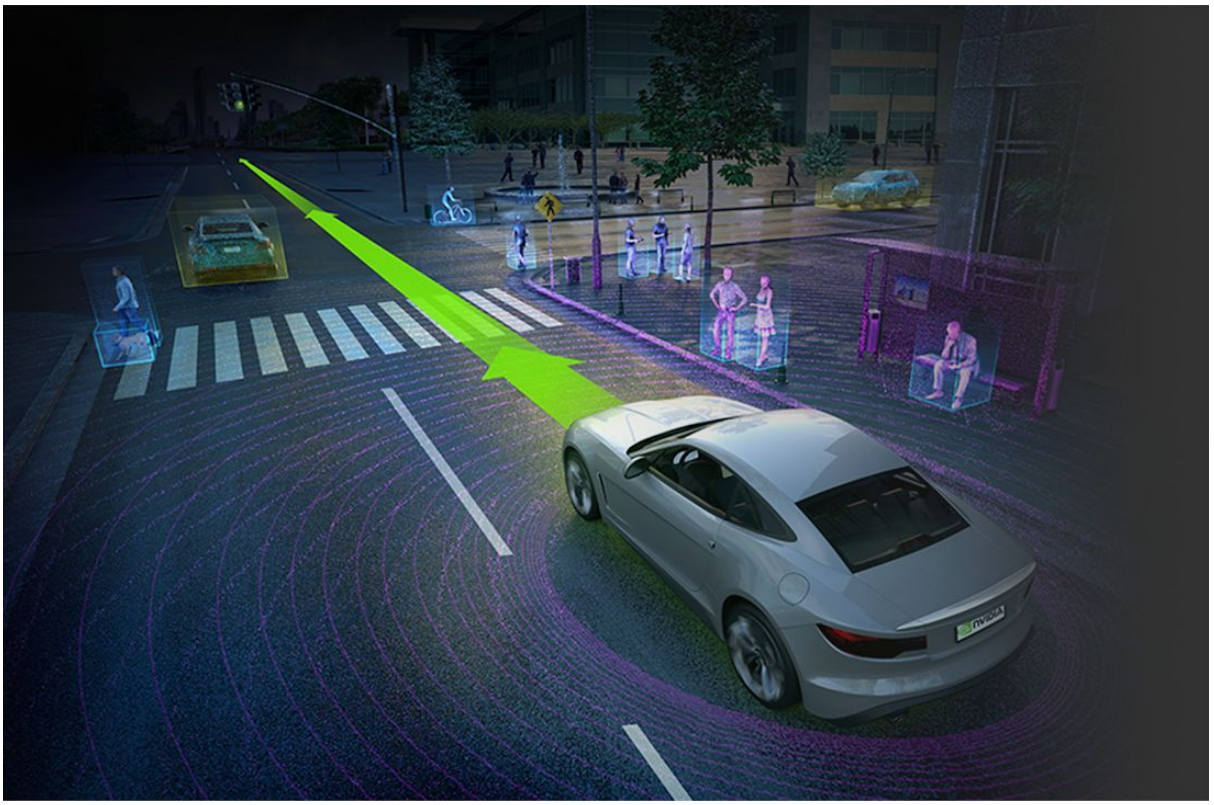
1. Teilnehmer

Vorname:	**Moritz**
Name:	**Wörmann**
Geb.-Datum:	**17.02.2003**
E-Mail:	**jugendforscht@mowoe.com**
Schule/Betrieb/Uni:	**Freiherr-vom-Stein-Gymnasium (Gymnasium)**

Kurzfassung:

Ziel dieses Projektes ist die Entwicklung eines selbstfahrenden Autos, welches nur per Camera-Vision autonom fährt. Außerdem soll das Auto mit verschiedenen Arten von Neuronalen Netzen selbst lernen zu fahren. Neuronale Netze sind kleine Programme, die den Lern-Prozess eines Menschen nachahmen und daher viele Probleme wie ein Mensch lösen können.

Selbstfahrende Autos mit neuronalen Netzen



Von Moritz Wörmann
mowoe@mowoe.com
Freiherr-vom-Stein Gymnasium
Münster

Inhaltsverzeichnis

1. Kurzfassung

Ziel dieses Projektes ist die Entwicklung eines selbstfahrenden Autos, welches nur per Camera-Vision autonom fährt. Außerdem soll das Auto mit verschiedenen Arten von Neuronalen Netzen selbst lernen zu fahren. Neuronale Netze sind kleine Programme, die den Lernprozess eines Menschen nachahmen und daher viele Probleme wie ein Mensch lösen können.

2. Einleitung

Da ich mich seit Längerem mit neuronalen Netzen beschäftige, war mir schnell klar, dass das Projekt etwas mit solchen zu tun haben sollte. Ich suchte dann nur noch einen geeigneten Anwendungsfall für neuronale Netze. Da stieß ich auf selbstfahrende Autos. Ich hielt dieses Thema für ein sehr interessantes, da hierzu noch vergleichsweise wenig erforscht wurde. Es gibt von verschiedenen Leuten verschiedene Ansätze, die gut funktionieren, ein gutes Beispiel sind die Tesla-Autos, die mit einem Autopiloten verkauft werden, oder auch die in den USA eingesetzten selbstfahrenden Taxis der Firma Uber.

3.Theorie

3.1 Neuronale Netze

Neuronale Netze sind mehr oder weniger ein Nachbau des menschlichen Gehirns und sollen dessen Lernprozesse nachahmen. Ein Netz besteht

meistens aus einem oder mehreren Layers. Die Werte gehen dann durch jedes Layer und werden dann in diesen weiterverarbeitet. Ein Layer beinhaltet mehrere Neuronen, die durch Synapsen verbunden sind. Diese Synapsen dienen als eine Art mathematischer Faktor. In der Synapse ist also ein Wert gespeichert, mit dem jeder Wert, der durch sie läuft, multipliziert wird.

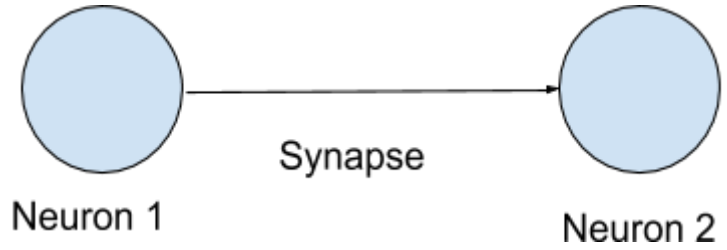


Abb. 1: Das einfachste "Neuronale Netz"

In Abb. 1 sieht man das einfachste Netz, welches nur aus zwei Neuronen und einer Synapse, die die beiden verbindet, besteht. Ein Beispiel-Problem, welches mit diesem Netz gelöst werden kann, ist Folgendes: Man hat wie bei einer Funktion x-Werte und y-Werte. Man möchte die notwendige Funktion aber nicht von Hand ausrechnen, sondern das neuronale Netz soll dies erledigen. Man hat z.B. diese Werte:

x	**y**
1	2
2	4
3	6

Tabelle 1: Beispielwerte für das Training eines neuronalen Netzes

Mit diesen Werten könnte man nun das neuronale Netz trainieren. Es wird also ein Wert gesucht, mit dem der x Wert multipliziert wird, um y zu erhalten. Für Menschen ist dies sehr einfach, man sieht auf den ersten Blick, dass der gesuchte Wert zwei ist. Aber dies könnte auch ein neuronales Netz errechnen. Der Trainingsprozess würde so ablaufen:

1. Das Neuronale Netz wird mit seinen Neuronen und Synapsen erstellt.
2. Für die Synapsen (in diesem Fall nur eine) wird ein Basiswert festgelegt, z.B. 1.
3. Dann wird die erste Iteration des Trainingsprozesses begonnen. Zuerst wird der x Wert einmal in das Netz eingespeist, um später den Fehler zu berechnen. Da der Faktor der Synapse im Moment noch 1 ist, würde bei dem ersten Wert 1 heraus kommen, da: 1 x 1 = 1. Dann wird

der Faktor der Synapse um einen festgelegten Wert erhöht. Dieser Wert kann z.B. 0.5 sein. Danach hat der Faktor den Wert 1.5. Nun wird der Fehler davor und danach errechnet, um zu prüfen, ob die Veränderung das Ergebnis positiv verändert hat. Die Fehler-Berechnung erfolgt folgendermaßen:

$$(Gewünschter\ Wert\ -\ Tatsächlicher\ Wert)^2\ =\ Fehler$$

Da man später nur wissen möchte, ob der Fehler sich vergrößert oder verkleinert hat, quadriert man den Fehler, um immer ein positives Ergebnis zu erhalten. Berechnet man also die beiden Fehler: $(2-1)^2=1$ und $(2-(1x1.5))^2=0.25$ erkennt man, dass die Veränderung gut war, da der Fehler kleiner geworden ist. Also behält man die Veränderung bei. Wäre der Fehler gleich geblieben oder größer geworden, würde der Trainingsprozess die Synapsen-Veränderung rückgängig machen und anstatt 0.5 zu addieren 0.5 subtrahieren und den Fehler erneut berechnen, anschliessend die Synapse entsprechend verändern oder gleich lassen.

Dieser Schritt 3 wird jetzt in einer Iteration für jeden x-Wert in der Tabelle wiederholt. Es werden so viele Iterationen durchlaufen wie vorher festgelegt wurden.

3.2 Aktivierungsfunktionen

Da meist die zu lernenden Funktionen nicht linear sind, benutzt man sogenannte Aktivierungsfunktionen, die dafür sorgen, dass das Netz nicht mehr linear ist. Diese Funktionen werden an die Neuronen gebunden, sodass jeder Wert, der durch das Neuron durch läuft, auch durch die Aktivierungsfunktion läuft. Im Folgenden sind Beispiele für Aktivierungsfunktionen gegeben:

Sigmoid:

Die Sigmoid-Funktion gibt immer einen Wert zwischen -1 und 1 zurück. Die Berechnung erfolgt folgendermaßen: $\frac{1}{1+e^{-10x}}$ oder $\frac{1}{1+e^{-5x}}$.

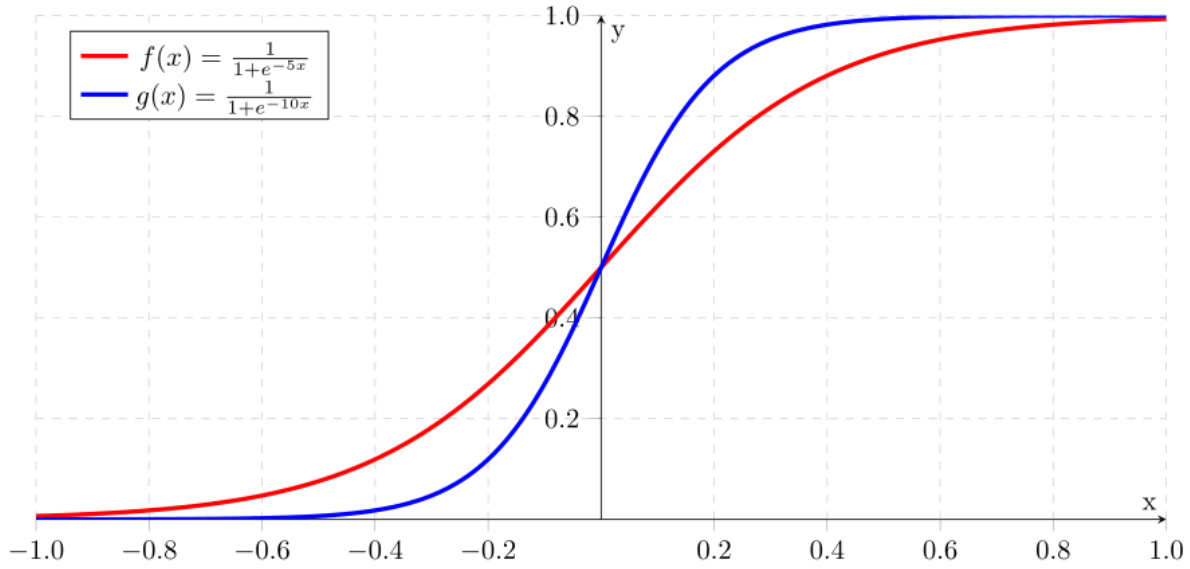


Abb.2: Der Graph der Sigmoid-Funktion

ReLU:

Die ReLU-Aktivierungsfunktion gibt 0 zurück, wenn die Zahl 0 oder kleiner ist oder wenn sie größer ist, die Zahl selbst. Diese Funktion wird in folgender mathematischer Schreibweise dargestellt :

$max(0, x)$

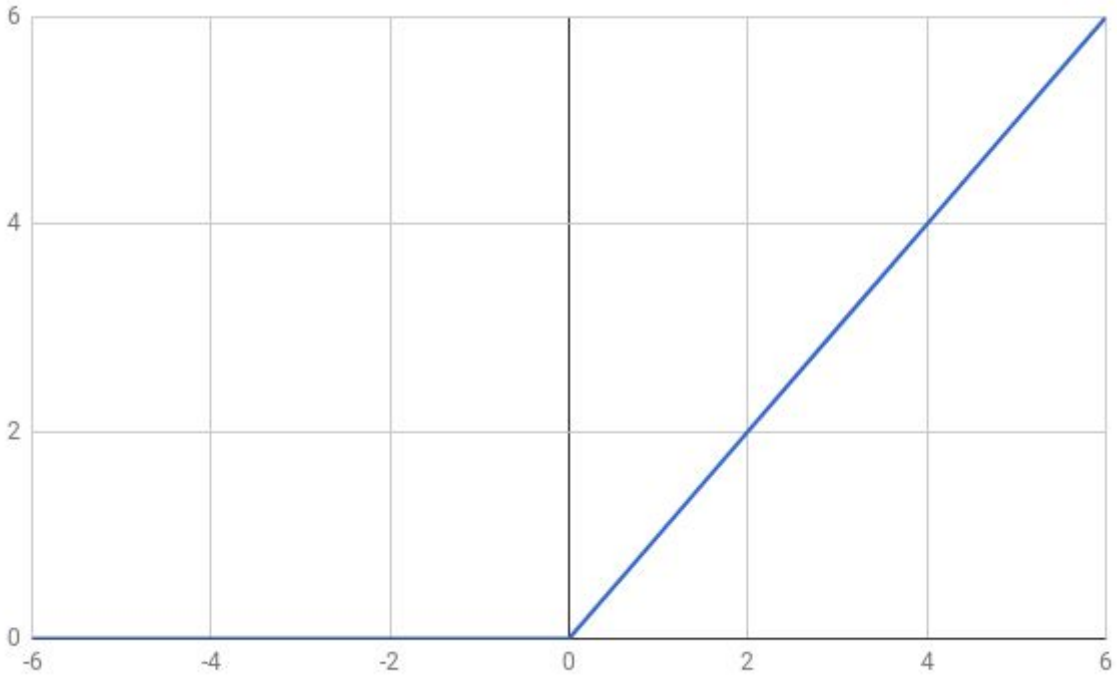


Abb.3: Der Graph der ReLU-Funktion

Tangens Hyperbolicus:
Der Tangens Hyperbolicus wird auch als Aktivierungsfunktion benutzt, da er nicht linear ist.

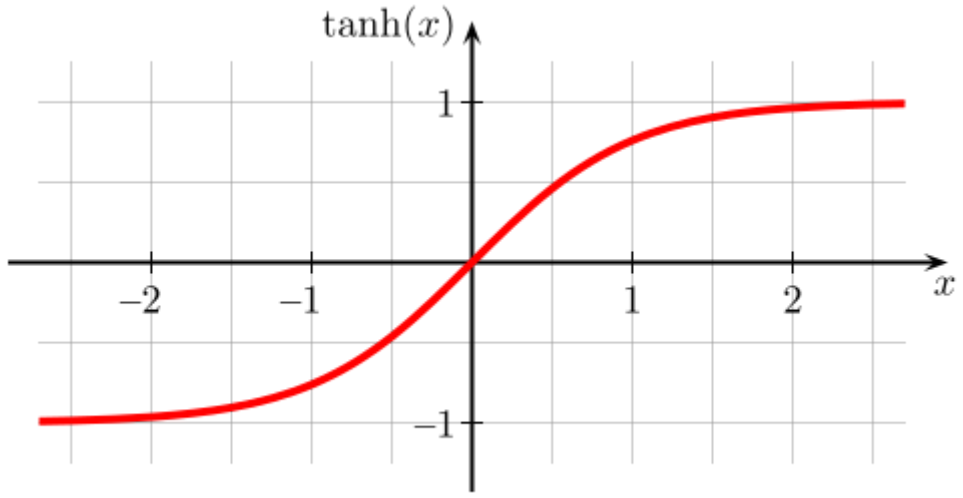


Abb.4: Der Graph von Tangens Hyperbolicus

3.3 Deep Neural Networks

Deep Neural Networks sind ähnlich wie normale Neuronale Netzwerke, wie beschrieben in 3.1., der Unterschied ist, dass sie mehrere Layer zwischen dem Anfangs- und End-Layer besitzen. Diese Layer werden auch Hidden Layer genannt. Ein Layer besteht aus mehreren Neuronen. Die Neuronen von zwei aufeinanderfolgenden Layers sind über sogenannte Synapsen verbunden. Über diese werden die Werte dann durch die Layer transportiert.

3.4 Keras und Tensorflow

Alle Programme aus den Versuchen sind in der Skriptsprache Python geschrieben. Für diese Sprache gibt es sogenannte Frameworks, die das Erstellen von Neuronalen Netzen ermöglichen und vereinfachen. Tensorflow ist von Google erstellt worden und Open-Source. Tensorflow wird mittlerweile in sehr vielen Unternehmen eingesetzt, da es ursprünglich für Android-Smartphones entwickelt wurde. Keras ist auch ein Framework, welches Tensorflow als Backend benutzt. Außerdem ist es etwas einfacher zu handhaben, schneller und einfacher. Aus diesem Grund wird Keras in diesem Projekt verwendet.

3.5 Layers in Keras

In Keras gibt es verschiedene vorgefertigte Layer, die in diesem Projekt benutzt werden. Hier sind sie erklärt:

Conv2D: Convolutional Layer: Ein Convolutional Layer besteht aus mehreren Neuronen wie in einem normalen Layer. Der Unterschied ist jedoch, dass jedes Neuron identisch ist. Außerdem ist ein Neuron nur mit ein paar Neuronen des vorherigen Layers verbunden. Da alle Neuronen in diesem Layer gleich sind, kann man sich diesen Layer wie ein kleines Raster, genannt *Kernel,* vorstellen, der über den Input-Vektor läuft.

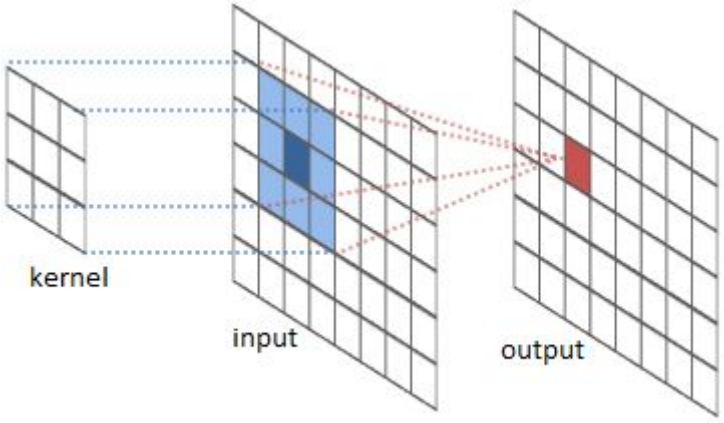


Abb.5: Visualisierung eines Convolutional Layers eines neuronalen Netzwerkes

Dieser Kernel besteht aus einer meist 3x3 oder 5x5 Werte großen Matrix. Wenn der Kernel also über ein Wert läuft, beachtet er auch z.B. alle 2 Werte um diesen Wert herum. Diese 9 oder 25 Werte werden dann mit den Faktoren aus dem Kernel multipliziert. Die Produkte dieser Multiplikationen werden addiert und ergeben dann den Output-Wert für den Pixel, über den der Kernel gerade läuft. Diese Layer werden meistens für Bild-Erkennung verwendet, da sie sehr gut geeignet sind, um Formen zu erkennen. Dies funktioniert, da Bilder als Vektoren dargestellt werden können. Hier noch ein Beispiel:

0	0	0	0	0
0	0	0	0	0
0	-1	1	0	0
0	0	0	0	0
0	0	0	0	0

Abb. 6: Ein Beispiel für ein Convolutional-Layer, um eine Ecken-Erkennung durchzuführen.

Mit dem Kernel links lässt sich eine sogenannte Edge detection durchführen, welche wie der Name sagt, Ecken auf einem Bild erkennt. Wenn also ein Pixel die Farbe Blau hat (RGB Wert: 255) und der Pixel rechts daneben auch Blau ist, dann ergibt die ganze Matrix 0:

0	0	0	0	0
0	0	0	0	0
0	0	-255	255	0
0	0	0	0	0
0	0	0	0	0

Tabelle 2: Ein Beispiel mit eingesetzten Werten für eine Ecken-Erkennung mithilfe eines Convolutional Layers

Die Summe aus diesen Werten ist null und deswegen ist an der Stelle der Pixel jetzt Schwarz, da hier keine Ecke erkannt wurde. Sind die beiden Farben nun aber nicht mehr gleich, beträgt die Summe nicht mehr null und ein Pixel mit der jeweiligen Farbe wird an der Stelle gezeichnet.

3.6 Training

Die Trainingsfunktionen verändern die Faktoren (Weights genannt) in den einzelnen Layers, im Prinzip genauso wie in 3.1. Die Trainingsfunktionen orientieren sich an den Ergebnissen einer Loss-Funktion. Diese berechnet den Fehler zwischen dem erhaltenen Ergebnis und dem gewünschten Ergebnis.

3.7 Canny Detection

Die Canny Detection funktioniert ähnlich wie ein Convolutional Layer. Ähnlich wie das Beispiel mit der Edge Detection. Eine Art Kernel wird über jedes Pixel des Bildes geschoben. Der neue Pixel ist die Summe aus allen Produkten der Multiplikationen der Werte aus dem Kernel und der Pixel um denjenigen Pixel, über den der Kernel gerade läuft. Die Kernel Matrix sieht so aus:

$$\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

Versuch 1:

Der erste Versuch war, ein selbstfahrendes Auto zu entwerfen, welches nur einen einzigen Algorithmus benötigt. Dies bedeutet ein Bild in den Algorithmus zu geben und ein Lenkrad Wert zu erhalten. Dieser Lenkrad-Wert sagt, wie weit das Lenkrad gedreht werden soll. Ein Vorteil ist, dass das Neuronale Netz beim Trainings-Prozess selbst heraus findet, was wichtig ist, um eine Lenkung zu berechnen. Ein Nachteil ist, dass man genau diese Faktoren nicht bestimmen und auch nicht später einsehen kann. Der komplette Algorithmus wird in Python geschrieben und Keras als Front-End und Tensorflow als Backend verwendet. Das Netz besteht aus 5 Convolutional-Layers und 5 Fully-Connected Layers. Zwischen diesen gibt es noch einen Flatten Layer (vgl. Abschnitt 3.5).

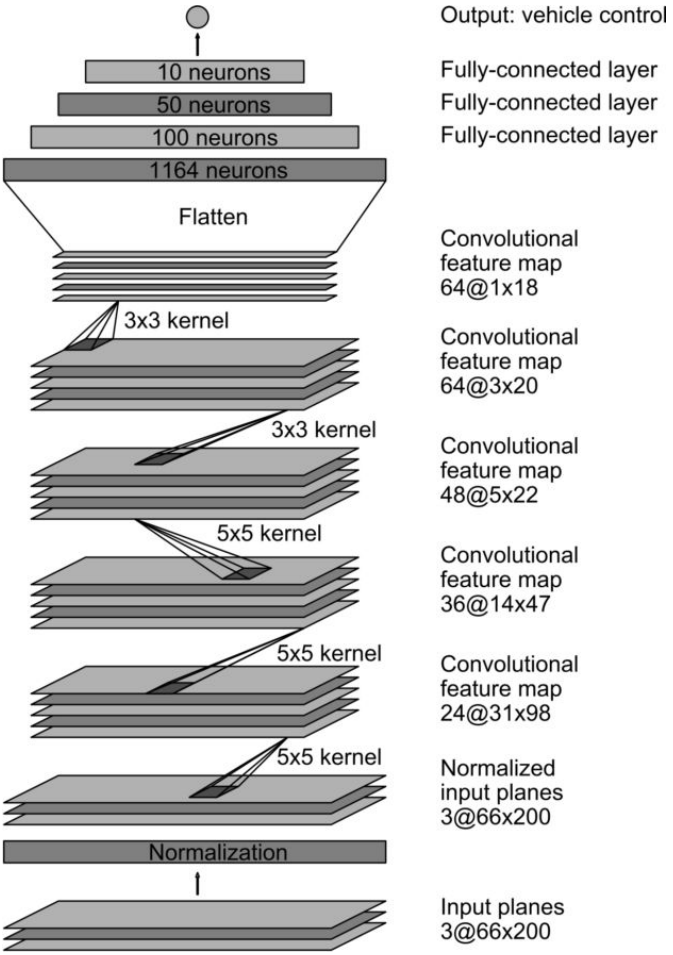


Abb. 7: Aufbau des Neuronalen Netzes für diesen Versuch

Generieren von Trainingsdaten:

Der Trainings-Datensatz wurde mit einem echten Auto erstellt. Dazu wurde eine Kamera in der Frontscheibe befestigt, welche die Straße filmt. Außerdem wurde ein Smartphone am Lenkrad befestigt. Dies wurde dazu verwendet, um die Lenkung aufzunehmen. Dafür wurde der Neigungssensor im Smartphone verwendet. Später wird dann der Neigungsgrad in eine Gradzahl umgerechnet. Im Rahmen des Versuch 1 wurde mit dem Auto einmal von Münster nach Düsseldorf über die Autobahn gefahren. Ein Problem war, dass die Zeit im Smartphone und im Computer nicht synchron waren und später erst die Bilder zu den Werten zugeordnet werden mussten. Ausserdem nahm ich an, um gute Ergebnisse zu erhalten, müssten die Bilder in einer verhältnisweise hohen Auflösung bleiben. Dies hat dazu geführt, dass das Training sehr viel Zeit in Anspruch genommen hat und auch viel Rechenleistung benötigt hat. Leider war die Genauigkeit nach den ersten Versuchen immer noch sehr gering. Dies ist entweder darauf zurückzuführen, dass die Bilder nicht mit den Werten übereinander lagen und ein Art Verschiebung vorlag, wodurch die Werte zu

den falschen Bildern zugeordnet wurden, oder es liegt ein Fehler im neuronalen Netz vor, diesen habe ich aber, wenn er existiert, noch nicht ausfindig machen können. Des weiteren ist es vermutlich auch ein guter Ansatz, die Trainings-Daten nicht zuerst mit einem echten Auto zu generieren. Stattdessen könnte man ein Simulationsprogramm für den Computer benutzen, da man hier deutlich einfacher Daten abgreifen könnte und deutlich mehr Trainingsdaten generieren könnte, um die Genauigkeit des neuronalen Netzes zu steigern.

Abb.8-9: Webcam in Windschutzscheibe eingebaut, um Fotos zu machen.

Abb.10: Befestigung des Smartphones am Lenkrad, um mit dem Neigungssensor die Drehung des Lenkrades aufzuzeichnen

Versuch 2:

Der zweite Versuch ist, die einzelnen Komponenten, die zum Lenken nötig sind, einzeln zu erkennen. Der erste Teil besteht aus der Straßenlinien-Erkennung. Dies wird jedoch nicht per Machine-Learning gelöst, da es deutlich bessere Methoden dazu gibt, wie im Folgenden erläutert wird:
Das folgende Bild zeigt den Blick aus der Windschutzscheibe:

Abb.11: Der Startpunkt der Straßenlinien-Erkennung, der Blick aus der Windschutzscheibe.

Dann wird das Bild zu Graustufen konvertiert, da Farben nicht relevant sind und den Algorithmus nur langsamer machen würden.

Abb. 12: Das Bild vom Anfang wurde nun zu Graustufen konvertiert.

Dann wird das Bild sozusagen “verwischt”, da Details unwichtig sind für die Linienerkennung.

Abb.13: Das Bild wurde “verwischt” da Details für eine Erkennung der Linien nicht notwendig sind

Dann wird eine sogenannte Canny-Detection durchgeführt (vgl. Abschnitt 3.7). Diese Canny-Detection versucht alle Ecken auf dem Bild zu erkennen.

Abb.14: Auf dem Bild wurden alle Ecken heraus gefiltert

Die Linien sind jetzt schon gut zu erkennen. Da die Linien nur in einem bestimmten Bereich des Bildes vorhanden sein können, wird jetzt ein Teil des Bildes ausgeschnitten:

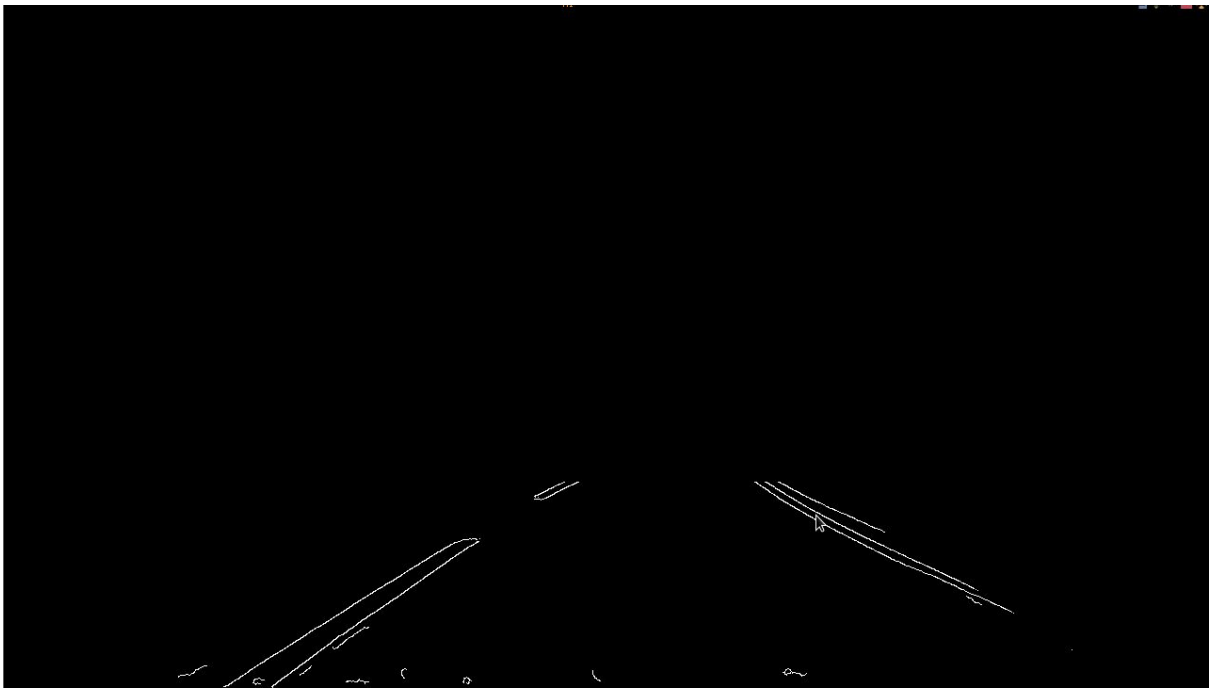


Abb.15: Ein Teil wird aus dem Bild ausgeschnitten, da nur in diesem Bereich die Linien vorhanden sind

Um jetzt die genauen Koordinaten der Linien auf dem Bild zu erhalten, wird eine so genannte Hough-Lines Detection durchgeführt. (vgl. Abschnitt 3.8).

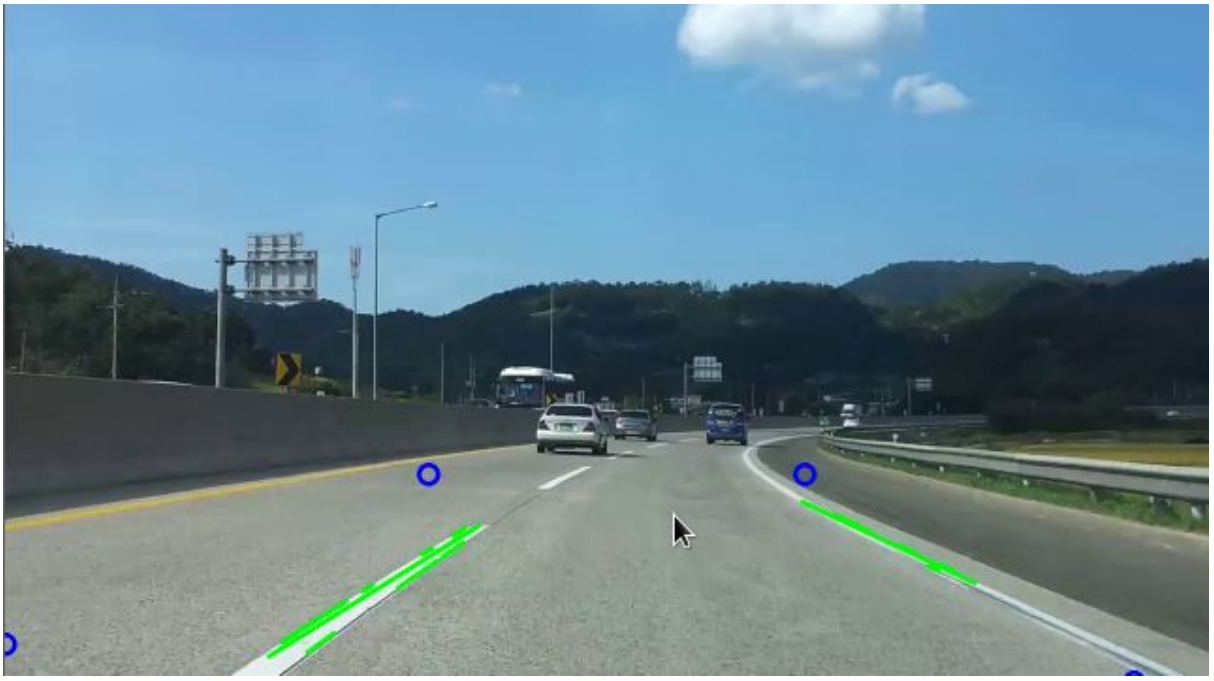


Abb.16:Nun wurden die Linien mit einer Hough-Transformation erkannt und in das Bild gezeichnet

Die blauen Punkte zeigen an, wo das Bild ausgeschnitten wurde. Die grünen Linien sind die von der Hough-Lines Erkennung erkannten Linien. Diese Linien werden jetzt noch per linearer Regression zu zwei einzelnen Linien zusammengefasst.

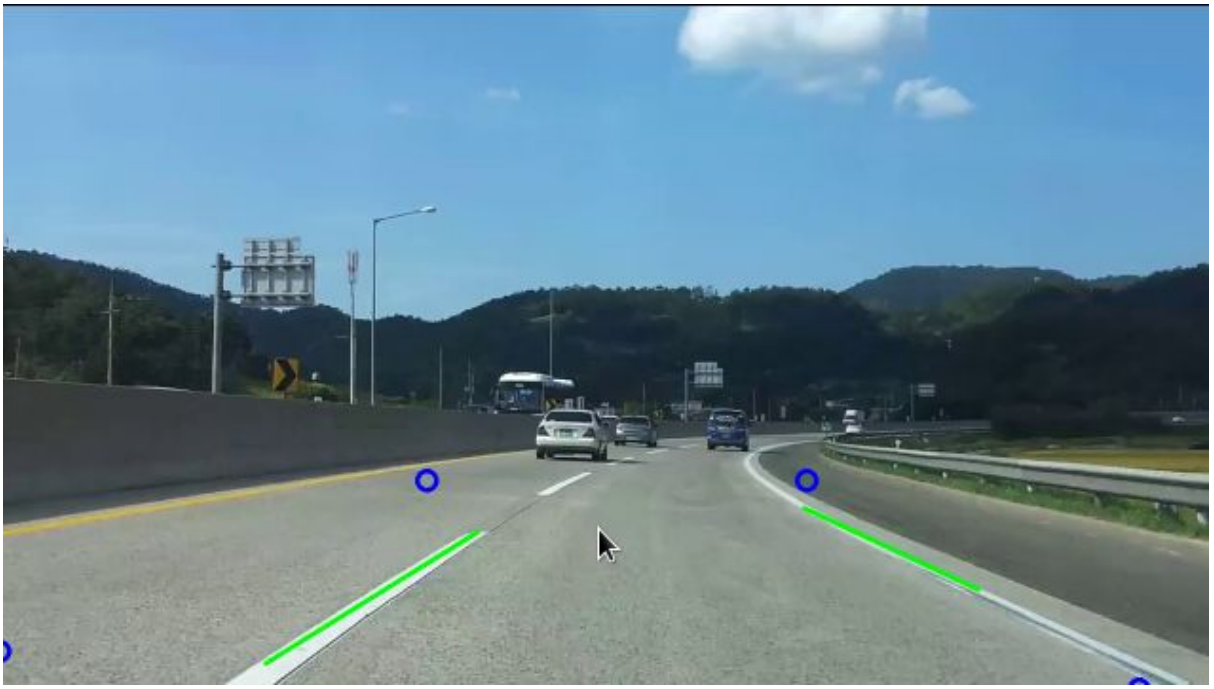


Abb.17: Die erkannten Linien wurden zu 2 einheitlichen Linien zusammen gefasst.

Fazit aus Versuch 2:

Es ist gelungen, eine Erkennung der Markierungslinien auf der Strasse zu erstellen. Jetzt könnte man die Ergebnisse in eine Lenkrad-Position umwandeln, um das Auto anhand der Straßen-Markierungen zu lenken. Weiter könnte man noch einzelne Algorithmen, die z.B. Stopp-Schilder oder Rote Ampeln erkennen können, hinzufügen, um die Steuerung des Autos noch auf mehr Faktoren aus zu weiten.

Fazit:

Ich habe versucht ein selbstfahrendes Auto zu bauen. Dies ist mir zum Teil gelungen. Ich habe einzelne Arten von Algorithmen ausprobiert. Als nächstes würde ich weitere Bausteine zum selbstfahrenden Auto entwickeln wie z.B. das erkennen und reagieren auf rote Ampeln und Stopp Schildern.

Literaturverzeichnis

Bild auf Deckblatt:
https://www.nvidia.de/content/dam/en-zz/de_de/Solutions/self-driving-cars/home/self-driving-car-main-banner-1279-d@2x.jpg

Abb.1: selbst erstellt
Abb.2:https://de.wikipedia.org/wiki/K%C3%BCnstliches_Neuron#/media/File:Sigmoid-function.svg
Abb.3: selbst erstellt

Abb.4:https://de.wikipedia.org/wiki/Tangens_hyperbolicus_und_Kotangens_hyperbolicus#/media/File:Hyperbolic_Tangent.svg
Abb.5:
https://colah.github.io/posts/2014-07-Understanding-Convolutions/img/RiverTrain-ImageConvDiagram.png
Abb.6:
https://colah.github.io/posts/2014-07-Understanding-Convolutions/img/Gimp-Edge.png
Abb.7:
https://devblogs.nvidia.com/parallelforall/wp-content/uploads/2016/08/cnn-architecture-624x890.png
Abb.8-17: selbst erstellt
Der Code für die einzelnen Versuche ist erhältlich auf: https://mowoe.com/jufo