

H T
W I
G N

Fakultät Informatik
Fachbereich Angewandte Informatik

Dokumentation Teamprojekt

**Autonomes Fahren in der DuckieTown-Umgebung -
Relative Posenschätzung und Linienverfolgung
mit einem Deep-Learning Ansatz**



Teilnehmer: Stephan Perren

Felix Mayer

Betreuer: Prof. Dr. Oliver Bittel

Konstanz, 23. Oktober 2020

Inhaltsverzeichnis

Inhaltsverzeichnis	II
Abkürzungsverzeichnis	III
1 Einleitung	1
1.1 Aufgabenstellung	2
2 DuckieTown	3
2.1 Umgebung und Simulator	3
2.2 DuckieBot	6
2.3 Bedienung	6
3 Deep Learning	8
3.1 Was ist Deep Learning?	8
3.2 Warum Deep Learning?	9
3.3 Lernverfahren	10
3.4 Datensatz	11
3.4.1 Trainingsdatensatz	11
3.4.2 Validierungsdatensatz	12
3.4.3 Testdatensatz	12
3.5 Funktionsweise und Aufbau künstlicher neuronaler Netze	12
3.5.1 Eingabeschicht (input layer)	12
3.5.2 Zwischenschichten (hidden layer)	13
3.5.3 Ausgabeschicht (output layer)	13
3.5.4 Modellparameter	13
3.5.5 Hyperparameter	13
3.5.6 Gewichte und Verzerrung (Bias)	13
3.5.7 Aktivierungsfunktion (activation function)	14
3.5.8 Verlustfunktion (loss function)	15
3.5.9 Fehlerrückführung (Backpropagation)	15
3.6 Multilayer-Perzeptron (MLP)	17
3.7 Convolutional Neural Networks (CNNs)	18
4 Vorgehensweise	19
4.1 Netzarchitektur	19
4.2 Datengewinnung	19
4.2.1 Label	21
4.2.2 Observationen	22
4.2.3 Datensätze	23

Inhaltsverzeichnis

4.3 Lernprozess	25
4.3.1 Hyperparameter	25
4.3.2 Anwendung	26
5 Ergebnisse	27
5.1 Training und Test	27
5.1.1 2D-Pose	27
5.1.2 1D-Pose	28
5.1.3 Steuerbefehl durch Expertensystem	29
5.2 Validierung	30
5.2.1 2D-Pose	30
5.2.2 1D-Pose	31
5.2.3 Steuerbefehl durch Expertensystem	32
6 Fazit	34
6.1 Rückblick	34
6.1.1 2D-Pose	34
6.1.2 1D-Pose	35
6.1.3 Steuerbefehl durch Expertensystem	35
6.2 Verbesserungen	35
6.2.1 2D-Pose	35
6.2.2 1D-Pose	36
6.2.3 Steuerbefehl durch Expertensystem	36
Abbildungsverzeichnis	37
Codeauflistung	38
Literaturverzeichnis	39

Abkürzungsverzeichnis

1D eindimensional

2D zweidimensional

3D dreidimensional

AIN Angewandte Informatik

CNN Convolutional Neural Network

CNNs Convolutional Neural Networks

ELU Exponential Linear Unit

GL Graphics Library

KI Künstlichen Intelligenz

LED Light-Emitting Diode

MAE Mean Absolute Error

MBE Mean Bias Error

MIT Massachusetts Institute of Technology

MLP Multilayer-Perzeptron

MSE Mean Squared Error

PD Proportional-Derivative

PID Proportional-Integral-Derivative

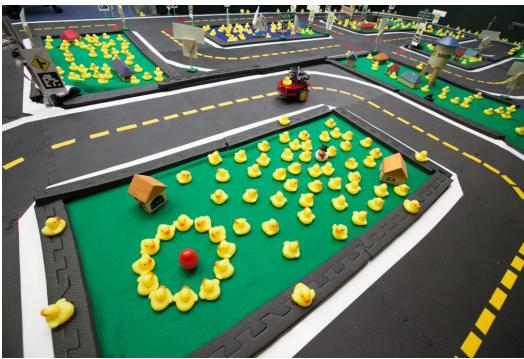
ReLU Rectified Linear Unit

RGB Rot-Grün-Blau (*Basisanteile eines Farbsignals*)

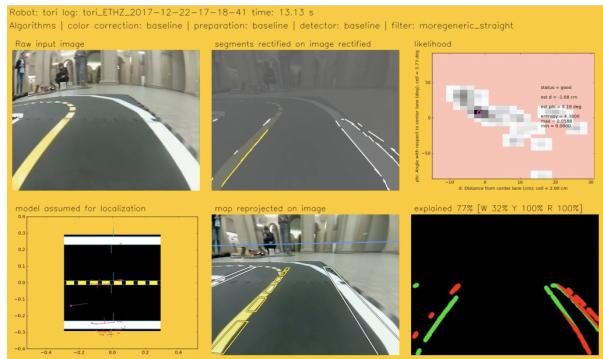
ROS Robot Operating System

WLAN Wireless Local Area Network

1 Einleitung



Quelle: https://www.duckietown.org/wp-content/uploads/2018/05/duckietown_nice-1024x683.jpg



Quelle: <https://www.duckietown.org/wp-content/uploads/2018/06/data-from-img-CameraDataProcessed-fc6fd822-1024x607.png>

Abbildung 1.1: DuckieTown

Das DuckieTown-Projekt wurde 2016 am Massachusetts Institute of Technology (MIT) konzipiert. Das Ziel war es, eine Plattform zu entwickeln, die klein, kostengünstig und “smart” ist, aber dennoch die wissenschaftlichen Herausforderungen einer echten autonomen Roboterplattform anbietet und die Entwicklung intelligenter autonomer Fahrfunktionen erlaubt. [1]

Ziel des AIN-Projekts ist die Implementierung verschiedener klassischer Robotik-Algorithmen für Navigation und Lokalisierung mit dem DuckieTown-Simulator

Nach einer Einarbeitung in die Fähigkeiten des Simulators sollen Algorithmen für die Linienvorfolgung, Lokalisierung mit einem Partikelfilter bei bekannter Umgebungskarte, Navigation (Planung, Wegeverfolgung und Hindernisvermeidung) realisiert werden. Optional können auch KI-Komponenten für Fahrverhalten und Erkennung von Verkehrszeichen entwickelt werden. Hierzu sollen neue Techniken wie Deep Neural Networks und Deep Reinforcement Learning zum Einsatz kommen.

Das Projekt, das für etwa 4-6 Personen in zwei bis drei Teams vorgesehen ist, soll in Python realisiert werden. Eventuell kommt das Robot Operating System (ROS) zum Einsatz.

1.1 Aufgabenstellung

Das Ziel unseres Teams ist die Auswertung der Kamerabilder eines Agenten in einer DuckieTown-Umgebung. Mit den daraus gewonnenen Informationen soll der Agent in der Lage sein, vollständig autonom die Fahrspur halten zu können. Benötigt wird daher ein System, welches anhand einer Kameraaufnahme Steuerbefehle für den Agenten berechnet.

Ein weiteres mögliches Ziel ist die Nutzung der Informationen zur Lokalisierung des Agenten. Über Odometrie kann die Orientierung relativ zur Startpose geschätzt werden. Weißt die Trajektorie über einen gewissen Zeitraum eine Änderung der Orientierung von etwa 90 beziehungsweise 270 Grad auf, kann davon ausgegangen werden, dass der Agent eine Kurve durchfahren hat. Ist außerdem die Umgebungskarte bekannt, kann mit Hilfe eines Monte-Carlo-Verfahrens der Agent global lokalisiert werden.

Wir werden zur Berechnung des Steuerbefehls auf zwei Ansätze eingehen:

- Inferenz der Pose des Agenten:

Aus der Aufnahme wird zunächst die relative Pose des Agenten zur Fahrspur inferiert. Diese besteht aus dem kürzesten Abstand d zu einer Straßenmarkierung, sowie aus der Winkeldifferenz θ zwischen Orientierung des Agenten und der Fahrbahn. Diese Werte werden dann mit einem PID-Regler konstant gehalten.

Dieser Ansatz hat den Vorteil, dass die Poseninformationen außerdem noch für die schnellere Lokalisierung des Agenten genutzt werden können, da das Durchfahren einer Kurve von dem Fahren von Schlangenlinien unterschieden werden kann, wenn die Steuerbefehle mit d abgeglichen werden.

- Direkte Inferenz des Steuerbefehls:

Der weiter verbreitete Ansatz der direkten Inferenz übergeht die Schätzung der Pose. Anhand der Daten wird direkt ein Steuerbefehl geschätzt, bestehend aus der Geschwindigkeit v und Winkelgeschwindigkeit ω . Da für einen optimalen Steuerbefehl mehr Informationen nützlich sind als d und θ , ist dieser Ansatz ebenfalls vielversprechend. So kann ein Expertensystem für Steuerbefehle beispielsweise den weiteren Verlauf der Spur berücksichtigen, konkret die Winkeldifferenz zwischen der aktuellen Orientierung des Agenten und der Orientierung der Fahrspur an einem kommenden Zielpunkt.

Der Abstand zur Seitenmarkierung d wird von uns als ein hundertstel einer Kachellänge des Simulators festgelegt. Diese Kachelgröße wiederum wird in der `yaml`-Datei der entsprechenden Karte festgelegt. Die Maße einer Kachel in der physikalischen Duckietown-Umgebung betragen 61cm^2 . Bei einer simulierten Kachelgröße von 1 und einem Fehler in d von 1 würde dies einem Fehler von 0.61 cm in der physikalischen Umgebung entsprechen.

Die Winkeldifferenz θ wird von uns in Grad im Wertebereich $[-180, 180]$ erfasst. Der Wertebereich des Steuerbefehls ist $[-1, 1]$ sowohl für v als auch ω . Diese Werte können später mit einem Faktor als Referenzgeschwindigkeit multipliziert werden.

2 DuckieTown

2.1 Umgebung und Simulator

Der DuckieTown-Simulator ist ein in `Python` und `OpenGL` beziehungsweise `Pyglet` geschriebener Simulator für das „DuckieTown-Universum“. Er basiert auf dem OpenAi Toolkit „Gym“ [2] und bietet die Möglichkeit DuckieBots (Agenten) in einer beliebigen DuckieTown-Umgebung zu platzieren und die Agenten darin zu navigieren. [3]

Der Simulator ist schnell, quelloffen und ausgesprochen anpassungsfähig. Er wurde zunächst für die einfache Linienverfolgung konzipiert und wurde dann im Laufe der Zeit zu einem voll funktionsfähigen Simulator für autonom fahrende Fahrzeuge, insbesondere im Bezug zur künstlichen Intelligenz. [3]

In einer DuckieTown-Umgebung wird die Umwelt für einen DuckieBot definiert. Ein DuckieBot kann diese Umgebung dann observieren und sich in dieser bewegen. Eine DuckieTown-Umgebung wird dabei aus verschiedenen Kacheln und Objekten aufgebaut. Eine beispielhafte DuckieTown-Umgebung ist in Abbildung 2.1 dargestellt.

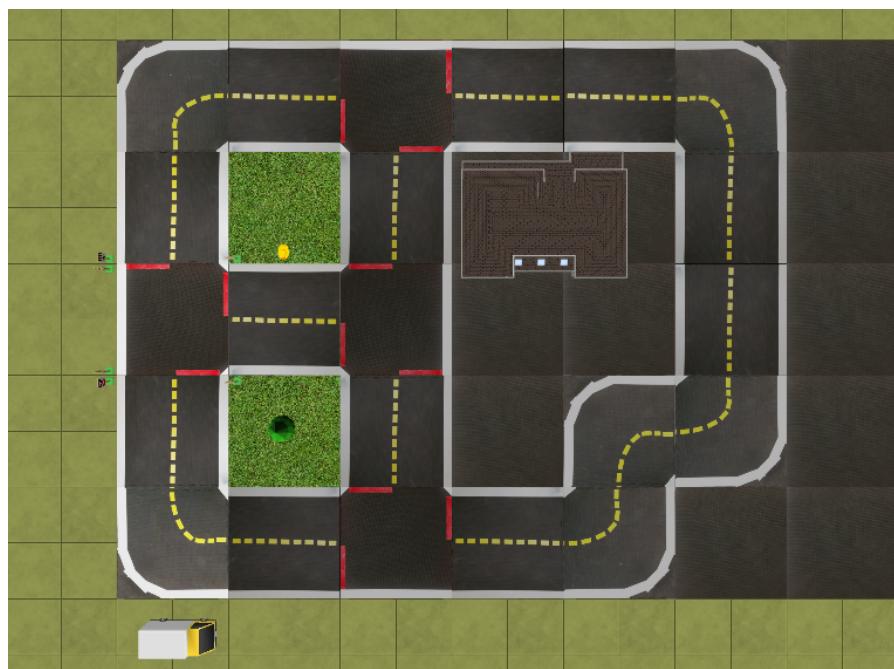


Abbildung 2.1: Darstellung einer beispielhaften DuckieTown-Umgebung

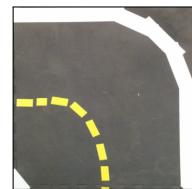
Quelle: <https://user-images.githubusercontent.com/10503729/45590954-c88c7c00-b912-11e8-9209-f72924684e21.gif>

Bei den Kacheln wird zwischen befahrbaren und nicht befahrbaren Kacheln unterschieden, wobei folgende Kacheln standardmäßig zur Verfügung stehen:

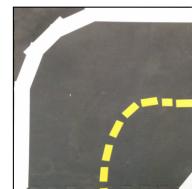
- Gerader Streckenabschnitt (befahrbar)



- Linkskurve (befahrbar)



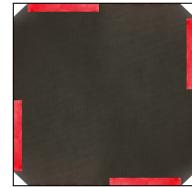
- Rechtskurve (befahrbar)



- 3-Wege Kreuzung (befahrbar)



- 4-Wege Kreuzung (befahrbar)



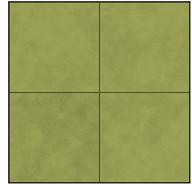
- Asphaltkachel (nicht befahrbar)



- Graskachel (nicht befahrbar)



- Bodenkachel (nicht befahrbar)



Objekte sind ebenfalls nicht befahrbar und befinden sich in der Regel außerhalb des Streckenverlaufes. Standardmäßig werden eine Vielzahl von Objekten zur Verfügung gestellt, wie zum Beispiel: Häuser, Bäume, Busse und Verkehrsschilder.

Eine DuckieTown-Umgebung wird hierbei in einer `.yaml`-Datei definiert (siehe Codebeispiel 2.1), die dann vom Simulator geladen und dargestellt wird .

Codebeispiel 2.1 Beispieldefinition einer DuckieTown-Umgebung

```
1  # 3x3 tiles with left turns at the corners
2  # going in a counter-clockwise loop
3
4  tiles:
5  - [curve_left/W , straight/W, curve_left/N]
6  - [straight/S , asphalt , straight/N]
7  - [curve_left/S , straight/E, curve_left/E]
8
9  tile_size: 1
```

2.2 DuckieBot

Ein DuckieBot ist ein kleiner mobiler Roboter, ausgestattet mit einem Raspberry Pi 3, einem Differenzialantrieb und einer Raspberry Pi Kamera. In der physikalischen Spezifikation besitzt der Roboter zur Kommunikation mit anderen Bots optional noch ein LED-Panel und WLAN-Adapter. Abbildung 2.2 zeigt die Darstellung eines physikalischen DuckieBots. Im Gym-DuckieTown Simulator existiert standardmäßig pro Umgebung nur ein Roboter.

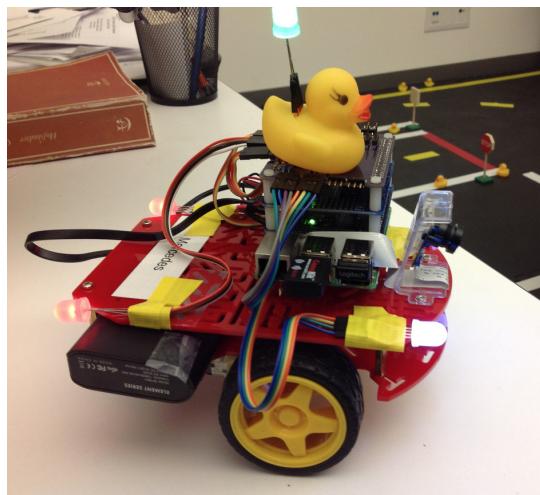


Abbildung 2.2: Darstellung eines DuckieBots

Quelle: <https://cdn-blog.adafruit.com/uploads/2016/05/mercedes.jpg>

2.3 Bedienung

Die Bedienung des Simulators erfolgt in einer einfachen Kontrollsleife. Zunächst wird eine DuckieTownEnv-Umgebung definiert und instanziert. Anschließend wird in einer

Schleife über die `step(action)` Methode der Agent gesteuert. Da die DuckieTownEnv-Umgebung einen Differenzialantrieb simuliert, ist `action` dabei ein Tupel aus Geschwindigkeit und Winkelgeschwindigkeit. Die Basisklasse `Simulator` dagegen nimmt zwei Werte als Dutycycle im Wertebereich -1 bis 1 für die beiden Antriebe entgegen.

Codebeispiel 2.2 Bedienung einer DuckieTown-Umgebung

```
1 env = DuckietownEnv()
2 obs = env.reset()
3 env.render()
4
5 while True:
6
7     lane_pose = get_lane_pos(env)
8     steering = k_p * lane_pose.dist + k_d * lane_pose.angle_rad
9     obs, reward, done, info = env.step(np.array([speed, steering]))
10    env.render()
11
12    if done:
13        env.reset()
```

Über die Methode `reset()` wird der Agent in einer zufälligen aber validen Pose platziert, also an einem befahrbaren und nicht durch andere Objekte besetzten Ort. Mit `render()` wird ein GL-Fenster geöffnet und ein Bild aus der Perspektive des Agenten angezeigt.

Die Methode `get_lane_pos(env)`, auch gerne „Cheat-Modul“ genannt, liefert Poseninformationen des Agenten relativ zur Fahrbahn. Diese bestehen aus dem Abstand zur „Ideallinie“, dem Abstand zur rechten Fahrbahnmarkierung, dem Skalarprodukt aus Richtungsvektor des Agenten und Tangente der Ideallinie am nächstgelegenen Punkt, sowie Winkeldifferenz (Arcuscosinus des Skalarprodukts).

Die Steuerung des Agenten erfolgt in diesem Beispiel über einen einfachen PD-Regler, wobei der Abstand zur Ideallinie `lane_pose.dist` als Fehler zum Sollwert interpretiert wird und die Differenz der Orientierungen von Agent und Fahrspur `lane_pose.angle_rad` als Ableitung des Fehlers.

Entgegengenommen wird der Steuerbefehl von der `step()` Methode des Simulators. Diese integriert den Befehl über eine definierte Zeitspanne `delta_time`, standardmäßig festgelegt als der Kehrwert der Bildrate des Simulators von 30 Bildern pro Sekunde. Zurück liefert die Methode eine RGB-Bildaufnahme aus Perspektive des Agenten (`obs`), einen Belohnungswert `reward` für das Halten der Spur und Vermeiden von Kollisionen, ein Flag `done`, welches anzeigt ob der Agent „verunglückt“ ist und neu platziert werden

muss, und das Wörterbuch `info` mit Zustandsinformationen des Agenten.

Das Ziel unseres Teams ist aus dem RGB-Bild `obs` den Wert `steering` zu inferieren.

3 Deep Learning

3.1 Was ist Deep Learning?

Unter Deep Learning (zu deutsch tiefes Lernen) versteht man ein Teilgebiet des maschinellen Lernens (siehe Abb. 3.1), welches sich mit künstlichen neuronalen Netzen und großen Datenmengen befasst. Es eignet sich für eine Vielzahl von Anwendungsfällen wie beispielsweise für selbstfahrende Autos, in der Medizin als auch im Marketing. [4]

Mit Deep Learning können Probleme gelöst werden, die ohne diese Ansätze nicht lösbar wären. Tiefes Lernen ist allerdings sehr rechenaufwändig, wodurch das Training über Monate hinweg andauern kann, um gute Entscheidungen treffen zu können. Gründe hierfür sind komplexe Architekturen sowie eine Vielzahl an Modell-Parametern. [4]

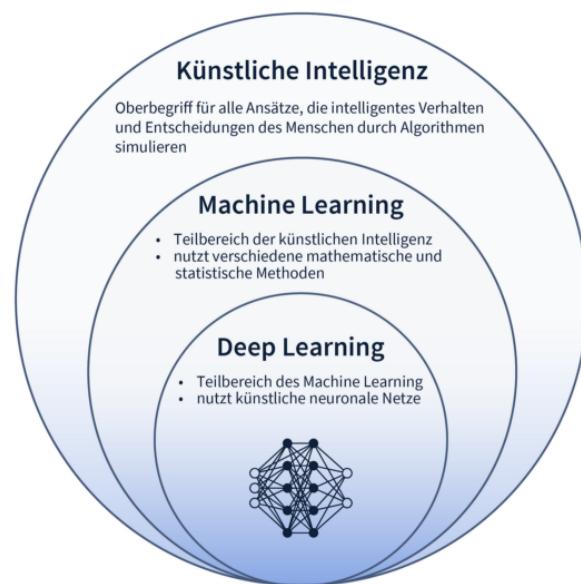


Abbildung 3.1: Übersicht Künstliche Intelligenz

Quelle: <https://datasolut.com/wp-content/uploads/2019/11/KI-und-Deep-Learning.002-e1558385989498.jpeg>

Die Grundlage des Deep Learnings stellt die Verwendung von künstlichen neuronalen Netzen dar. Unter künstlichen neuronalen Netzen versteht man Algorithmen, die nach

dem biologischen Vorbild des menschlichen Gehirns modelliert sind. Diese werden eingesetzt, um beispielsweise Muster in Bildern zu erkennen oder Bilder zu klassifizieren. [4]

Ein einfaches künstliches neuronales Netz besteht dabei aus einer Eingabeschicht (Input Layer), einer Zwischenschicht (Hidden Layer) und einer Ausgabeschicht (Output Layer) [4]. Abbildung 3.2 zeigt eine einfache Darstellung eines beispielhaften künstlichen neuronalen Netzes.

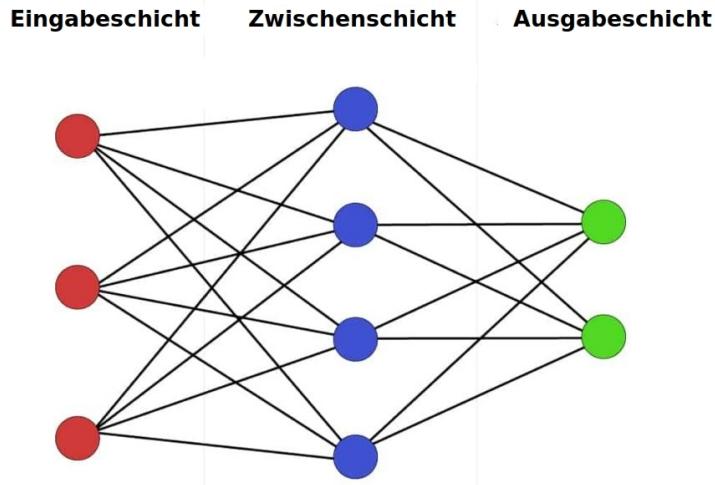


Abbildung 3.2: Darstellung eines beispielhaften künstlichen neuronalen Netzes
(vereinfacht)

Quelle: <https://datasolut.com/wp-content/uploads/2019/10/ku%CC%88nstliche-neuronale-Netze.jpg>

Von tiefem Lernen spricht man dann, wenn die eingesetzten neuronalen Netze mehr als eine Zwischenschicht haben. [4]

3.2 Warum Deep Learning?

Es gibt Problemstellungen (wie beispielsweise die unstrukturierte Bilderkennung), die sich besonders gut mit künstlichen neuronalen Netzen lösen lassen. Das Erlernen dieser komplexen Muster ist jedoch mit klassischen Machine Learning Algorithmen nur sehr schwer lösbar. Hier kommen dann tiefe künstliche neuronale Netze zum Einsatz. Je größer die Datenmenge ist, die zum Lernen verwendet wird, desto besser funktioniert das tiefe Lernen [4]. In Abb. 3.3 ist die Performance von Deep Learning Algorithmen im Vergleich zu älteren Lernalgorithmen dargestellt.

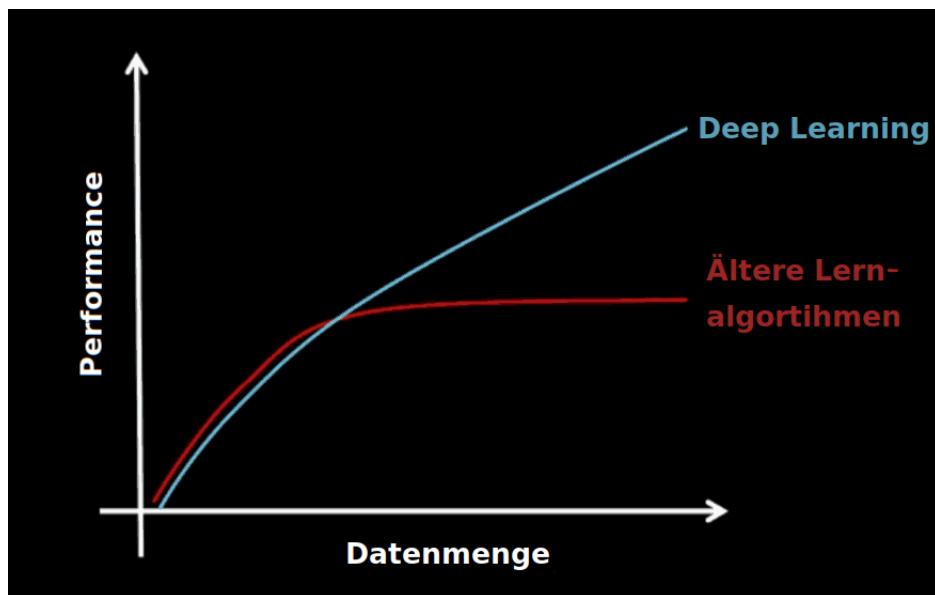


Abbildung 3.3: Darstellung der Performance von Deep Learning Algorithmen im Vergleich zu älteren Lernalgorithmen

Quelle: <https://datasolut.com/wp-content/uploads/2019/11/Warum-deep-learning-1024x742.png>

3.3 Lernverfahren

Beim maschinellen Lernen beziehungsweise beim Deep Learning stehen drei unterschiedliche Lernverfahren zur Verfügung:

- Überwachtes Lernen (Supervised Learning)

Das überwachte Lernen nutzt für den Lernprozess bekannte Daten, um daraus Muster und Zusammenhänge zu erkennen. Die Muster werden anhand eines Trainingsdatensatzes (Beispieldaten) erlernt. Dabei wird der Zusammenhang zu einer Zielvariable erlernt und es wird versucht diese richtig vorherzusagen. [5]

- Unüberwachtes Lernen (Unsupervised Learning)

Das unüberwachte Lernen nutzt für den Lernprozess keine Beispieldaten, sondern Rohdaten, aus denen eigenständig Muster erkannt werden sollen.

Der hauptsächliche Unterschied zum überwachten Lernen ist also, dass das unüberwachte Lernen nicht dafür ausgelegt ist, eine Vorhersage für eine bekannte Zielvariable zu treffen. [5]

- Bestärkendes Lernen (Reinforcement Learning)

Anders als das überwachte beziehungsweise unüberwachte Lernen nutzt das bestärkende Lernen zunächst keine Daten, sondern diese entstehen in einer Simulationsumgebung nach einem Versuch-und-Irrtum-Verfahren. [6]

In Abbildung 3.4 ist eine Übersicht über die verschiedenen Lernverfahren dargestellt.



Abbildung 3.4: Darstellung der verschiedenen Lernverfahren

Quelle: <https://1.bp.blogspot.com/-xstYqLb90Bw/XSYZUUDOWrI/AAAAAAAAML4/4sxvpGjIsDgmR7bDYyhcKdfM0TbvIIHdwCEwYBhgL/s400/machine-learning-lernverfahren.png>

Welches Lernverfahren schlussendlich eingesetzt wird, hängt hierbei vom Anwendungsfall ab sowie mit den daraus resultierenden Datensätzen.

3.4 Datensatz

Damit ein künstliches neuronales Netz mit überwachtem Lernen, seine Aufgabe erfüllen kann, muss zunächst ein Datensatz erstellt werden, damit das Netz trainiert werden kann. Der Datensatz besteht dabei aus einem Trainingsdatensatz, einem Validierungsdatensatz sowie aus einem Testdatensatz.

3.4.1 Trainingsdatensatz

Der Trainingsdatensatz ist ein Beispieldatensatz der für das Lernen der Muster und Zusammenhänge in den Daten verwendet wird. Das Modell des neuronalen Netz nutzt also diese Daten um zu lernen. [7]

3.4.2 Validierungsdatensatz

Der Validierungsdatensatz ist ebenso ein Beispieldatensatz, welcher für die Abstimmung der Hyperparameter des Modells des neuronalen Netzes verwendet wird. Dadurch wird das sogenannte „Overfitting“ (Überanpassung) des Modells auf die Trainingsdaten verhindert. [7]

3.4.3 Testdatensatz

Der Testdatensatz ist ebenfalls ein Beispieldatensatz, jedoch sind die Daten von den Trainingsdaten unabhängig. Die Testdaten werden beim Training des neuronalen Netzes nicht benutzt, sondern dienen zur abschließenden Verifikation des Modells. Dadurch kann die Qualität des Modell erfasst werden, damit man eine Aussage über die Leistungsfähigkeit des neuronalen Netzes treffen kann. [7]

3.5 Funktionsweise und Aufbau künstlicher neuronaler Netze

Die Neuronen (Knotenpunkte) eines künstlichen neuronalen Netzes sind schichtweise angeordnet. Diese werden normalerweise in einer festen Hierarchie miteinander verbunden. Die Neuronen sind dabei im Regelfall zwischen zwei Schichten verbunden, jedoch in Ausnahmefällen aber auch innerhalb einer Schicht. [8]

Die Informationen fließen beginnend mit der Eingabeschicht über eine oder mehrere Zwischenschichten bis hin zur Ausgabeschicht. Dabei ist der Ausgabewert eines Neurons der Eingabewert des nächsten Neurons. [8]

Künstliche neuronale Netze werden meist schematisch horizontal dargestellt (wie z.B. in Abbildung 3.2). Die Eingabeschicht befindet sich dabei auf der linken Seite gefolgt von den Zwischenschichten in der Mitte und der Ausgabeschicht auf der rechten Seite. Die Anzahl der Schichten die in einem künstlichen neuronalen Netz verwendet werden, ist eine wichtige beschreibende Information. Enthält ein Netz zum Beispiel drei Schichten, so spricht man von einem drei-schichtigen neuronalen Netz. [8]

3.5.1 Eingabeschicht (input layer)

Die Eingabeschicht definiert den Startpunkt des Informationsflusses in einem künstlichen neuronalen Netz. Die Eingangssignale werden dabei von den Neuronen am Anfang dieser Schicht entgegengenommen und am Ende gewichtet an die Neuronen der ersten Zwischenschicht weitergegeben. [8]

3.5.2 Zwischenschichten (hidden layer)

Zwischen der Eingabe- und der Ausgabeschicht befindet sich mindestens eine Zwischenschicht. Je mehr Zwischenschichten ein künstliches neuronales Netz besitzt, desto tiefer ist das Netz, jedoch bewirkt jede weitere hinzukommende Zwischenschicht auch einen Anstieg der benötigten Rechenleistung. [8]

3.5.3 Ausgabeschicht (output layer)

Die Ausgabeschicht bildet die letzte Schicht eines künstlichen neuronalen Netzes, wobei diese sich hinter den Zwischenschichten befindet. Sie stellt das Ende des Informationsflusses eines künstlichen neuronalen Netz dar und enthält das Ergebnis der Informationsverarbeitung. [8]

3.5.4 Modellparameter

Die Modellparameter eines künstlichen neuronalen Netzes werden beim Training des Netzes selbstständig erlernt. Basierend auf bereits vorhandenen Parametern trainiert der Trainingsalgorithmus und versucht dadurch, die Modellparameter immer weiter zu verbessern. [9]

3.5.5 Hyperparameter

Die Hyperparameter eines künstlichen neuronalen Netz dienen zur Steuerung des Trainingsalgorithmus. In der Regel müssen die Werte der Hyperparameter im Gegensatz zu den Modellparametern vor dem eigentlichen Training des Modells festgelegt werden, wodurch diese nicht selbstständig erlernt werden können. [10]

Einige Beispiele für Hyperparameter sind:

1. Lerngeschwindigkeit (learning rate):

Schrittgröße in Richtung Minimum der Fehlerfunktion pro Iteration

2. Batch Size:

Anzahl an Beispielen welche pro Iteration in das Netz eingespeist werden

3. Anzahl Epochen:

Anzahl der Trainingsdurchläufe über den Datensatz

3.5.6 Gewichte und Verzerrung (Bias)

Durch Gewichte wird die Intensität des Informationsflusses entlang einer Verbindung eines künstlichen neuronalen Netzes beschrieben. Dazu vergibt jedes Neuron ein Gewicht

für die durchfließende Information. Diese wird anschließend mit diesem Gewicht gewichtet und es wird ein Wert für die neuronen-spezifische Verzerrung (Bias) addiert. Das resultierende Ergebnis wird in der Regel durch eine sogenannte Aktivierungsfunktion geleitet, bevor es an die Neuronen der nächsten Schicht weitergegeben wird. [8]

Während des Trainingsprozesses werden die Gewichte und Verzerrungen so angepasst, dass das Endresultat möglichst genau den Anforderungen entspricht. [8]

3.5.7 Aktivierungsfunktion (activation function)

Aktivierungsfunktionen spielen in künstlichen neuronalen Netzen eine bedeutende Rolle, da sie dabei helfen, die komplizierte und nichtlineare funktionale Beziehung zwischen den Eingangsdaten und den abhängigen Ergebnissen zu lernen und zu verstehen. Ein Eingangssignal eines Neurons wird dabei in ein Ausgangssignal konvertiert, welches anschließend als Eingabe der nächsten Schicht verwendet wird. [11]

Einige Beispiele für Aktivierungsfunktionen sind:

Bezeichnung	Funktion	Plot
Identität	$f(x) = x$	
Sigmoid	$f(x) = \frac{1}{1+e^{-x}}$	
ReLU	$f(x) = \max(0, x)$	
ELU	$f(\alpha, x) = \begin{cases} \alpha(e^x - 1) & \text{für } x \leq 0 \\ x & \text{für } x > 0 \end{cases}$	

3.5.8 Verlustfunktion (loss function)

Die Verlustfunktion wird zur Berechnung des Fehlers zwischen den realen Ergebnissen und erhaltenen Antworten des künstlichen neuronalen Netzes verwendet. Das Ziel des Lernprozesses ist die Minimierung dieser Fehler. [11]

Einige Beispiele für Verlustfunktionen sind:

Bezeichnung	Funktion
Mittlere quadratische Abweichung	$MSE = \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{n}$
Mittlerer absoluter Fehler	$MAE = \frac{\sum_{i=1}^n y_i - \hat{y}_i }{n}$
Mittlerer Bias Fehler	$MBE = \frac{\sum_{i=1}^n (y_i - \hat{y}_i)}{n}$

3.5.9 Fehlerrückführung (Backpropagation)

Bei der Fehlerrückführung (Backpropagation) wird die Antwort des künstlichen neuronalen Netzes mit dem gewünschten Ergebnis verglichen. Dabei wird der Fehler bestimmt und dieser wird anschließend rückwärtig in das Netz gespeist, wodurch die Gewichte der Neuronen so angepasst werden, dass der Fehler immer kleiner wird. In der Praxis werden dafür verschiedene Optimierungsstrategien eingesetzt. [12]

3.5.9.1 Optimierungsstrategien (Optimizer)

1. Gradientenverfahren:

Das Gradientenverfahren beschreibt einen Algorithmus, der für eine Verlustfunktion, den Eingabeparameter in entgegengesetzter Richtung des Gradienten aktualisiert. Über die Lerngeschwindigkeit (learn rate) kann festgelegt werden, wie weit der Algorithmus sich in jeder Iteration vom Ausgangspunkt entfernen darf. [13]

2. Momentum Optimierer:

Der Momentum Optimierer ist ein erweiterter Ansatz des Gradientenverfahrens. Während das „Standardgradientenverfahren“ in jedem Iterationsschritt immer in entgegengesetzte Richtung des lokalen Gradienten absteigt, nimmt der Momentum Optimierer Rücksicht auf die Gradienten der vorherigen Iterationen. [13]

3. Adaptive Optimierungsstrategie:

Ein Abstieg wie ihn die Gradientenverfahren erledigen, kann unter Umständen nicht die richtige Wahl zur Lösung des Problems darstellen. Daher wurden adaptive Algorithmen entwickelt, die eine Alternative zu den Gradientenverfahren darstellen. Dabei wird folgende Idee verfolgt: Man passt die Lernrate dynamisch an die Parameter an und führt größere Aktualisierungen für seltene Parameter durch und kleinere Aktualisierungen für häufige Parameter. [13]

a) **AdaGrad:**

Der Kerngedanke hinter AdaGrad ist das Herunterskalieren des Gradientenvektors, das heißt die Lernrate wird pro Dimension nach und nach verringert, wobei die Verringerung bei steilen Dimensionen schneller erfolgt. [13]

b) **RMSProp:**

RMSProp ist eine leicht modifizierte Version des AdaGrad Optimierers. Dabei werden die Gradienten der letzten Iterationen unter exponentiellem Zerfall berücksichtigt. Im Allgemeinen lässt sich sagen, dass RMSProp schneller konvergiert als AdaGrad. [13]

4. Adam Optimierung:

Die Adam Optimierung ist eine Kombination aus der Idee des Momentum Optimierers und RMSProp. Das Verfahren betrachtet hierbei sowohl den Durchschnitt der vorigen Gradienten als auch den Durchschnitt der quadrierten Gradienten, beide unter exponentiellem Zerfall. [13]

3.5.9.2 Faltungsschicht (convolution layer)

Arbeitet man mit Daten mit großer Dimensionalität, wie zum Beispiel Bildern, wird das Training mit vollständig verbundenen Netzen schnell unpraktikabel. Allein die Kombination eines ein-megapixel großen Bildes mit einer Schicht aus tausend Neuronen würde eine Milliarde Modellparameter ergeben. Es bietet sich daher das Arbeiten mit der Faltung an, wobei nur ein Faltungskern erlernt werden muss.

3.5.9.3 Dropout

Eine weit verbreitete Methode um das Überstimmen (Overfitting) von neuronalen Netzen zu minimieren sind Dropout-Schichten. Die Idee ist, dass ein für das Netz wichtiges Merkmal nicht aus der Aktivierung eines einzelnen Neurons besteht, sondern aus der Kombination mehrerer. Um die Abhängigkeit des Outputs des Netzes von einzelnen Aktivierungen zu verhindern, werden diese mit einer festgelegten Wahrscheinlichkeit auf null gesetzt. Dies geschieht nur während des Trainings und wird zur Inferenz abgeschaltet.

3.5.9.4 Normalisierung

Die Normalisierung der Input-Daten gilt weithin als Best-Practice. Der wichtigste Grund dafür ist schnellere Konvergenz und damit ein schnellerer Lernprozess. Dies lässt sich damit erklären, dass das Lernverfahren als eine Minimumssuche auf der Oberfläche der Fehlerfunktion verstanden werden kann, wobei die Größenordnung der Input-Daten die Verzerrung dieser Oberfläche bestimmen. Da das Netz den Wert der Fehlerfunktion

mit seinen Parametern minimieren soll, müssen bei unterschiedlich großen Inputs auch die Gewichte unterschiedlich groß sein. Das wiederum zieht unterschiedlich große ideale Lernraten nach sich. Diese Verzerrung der Oberfläche der Fehlerfunktion kann durch Normalisierung der Daten minimiert werden.

3.5.9.5 Regularisierung

Neuronale Netze bilden den Input über ihre Gewichte auf einen gewissen Output ab. Ist die Größenordnung mancher Gewichte größer als die von anderen, ist die Wahrscheinlichkeit hoch, dass das Netz überstimmt ist. Eine kleine Abweichung im Input kann dann drastische Auswirkungen auf den Output haben. Um zu verhindern, dass der Lernprozess die Gewichte beliebig groß gestaltet, kann an die Fehlerfunktion ein zusätzlicher Term angefügt werden, welcher die Größe der Gewichte beinhaltet. Eine L2-Regularisierung zum Beispiel, multipliziert das Quadrat der Größe der Gewichte mit einem Vorfaktor, einem Hyperparameter, und addiert das Ergebnis auf den Wert der Fehlerfunktion. Große Gewichte werden somit bestraft.

3.6 Multilayer-Perzeptron (MLP)

Ein einzelnes Perzeptron kann für sich genommen lediglich eine lineare Klassifikation erreichen (siehe Abb. 3.5i) Es ist jedoch des Öfteren notwendig, weitaus komplexere Klassifikationen zu bewerkstelligen. Dies lässt sich erreichen, indem mehrere Perzeptronen miteinander verschaltet werden. Dabei werden die einzelnen Perzeptronen zunächst in Schichten angeordnet. Die Perzeptronen einer Schicht, sind dann jeweils mit jedem Perzeptron der folgenden Schicht verschaltet. Innerhalb einer Schicht werden die Perzeptronen nicht miteinander verbunden und es darf auch keine zyklischen Verschaltungen zwischen den Schichten geben. Das resultierende Ergebnis wird dann als sogenanntes Multilayer-Perzeptron (MLP) bezeichnet. Die erste Schicht ist hierbei die Eingabeschicht. Sie ist dafür zuständig, die einzelnen Werte der Eingänge über die Eingabeperzeptronen an jedes Perzeptron der nächsten Schicht weiter zu leiten. Nach der Eingabeschicht folgt eine beziehungsweise mehrere Zwischenschichten. In den Zwischenschichten findet dann der eigentliche Klassifikationsprozess statt. Die nichtlineare Klassifikation (siehe Abb. 3.5ii) entsteht dabei durch den Einsatz von mehreren Perzeptronen. Nach den Zwischenschichten folgt letztendlich die Ausgabeschicht, welche aus den sogenannten Ausgabeperzeptronen besteht. Das Ergebnis des Klassifikationsprozesses wird dann schlussendlich über diese Schicht ausgegeben. Jede Schicht des MLP kann dabei aus mehreren Perzeptronen bestehen, aber es muss immer mindestens ein Perzeptron vorhanden sein. [14]

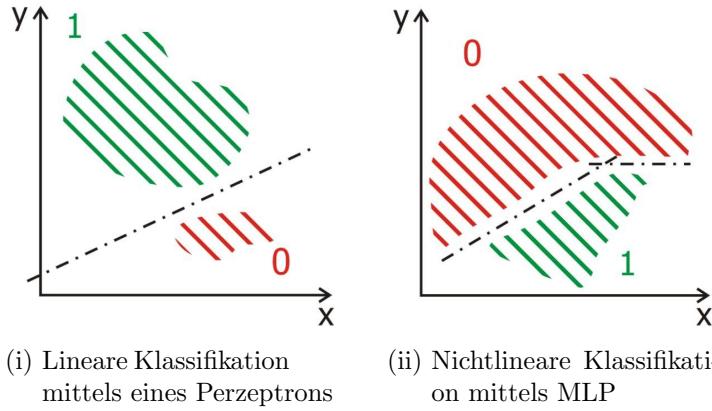


Abbildung 3.5: Lineare Klassifikation versus nichtlineare Klassifikation
Quelle: <https://www.informatik.uni-ulm.de/ni/Lehre/WS04/ProSemNN/pdf/MLP.pdf>

3.7 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) können besonders effizient mit 2D- beziehungsweise 3D-Eingabedaten arbeiten. Insbesondere werden CNNs für die Objektdetektion in Bildern angewendet. [15]

In Abbildung 3.6 kan man die Darstellung eines beispielhaften Convolutional Neural Networks sehen.

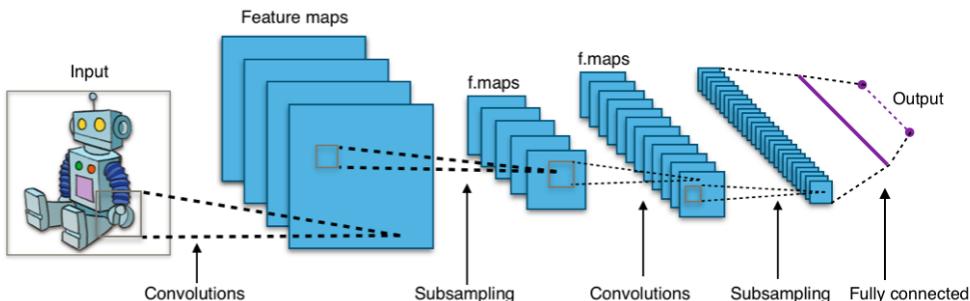


Abbildung 3.6: Darstellung eines beispielhaften CNN

Quelle: https://de.wikipedia.org/wiki/Convolutional_Neural_Network#/media/Datei:Typical_cnn.png

Der Unterschied zu den klassischen künstlichen neuronalen Netzen liegt in der Architektur der CNNs: Die Zwischenschicht basiert hierbei auf einer Abfolge von Faltungs- und Poolingoperationen. Bei der Faltung wird ein sogenannter Faltungskernell über die Daten geschoben und das Ergebnis der Faltungsoperation wird berechnet. Anschließend werden die Neuronen aktualisiert. Die nachfolgende Poolingoperation sorgt dann dafür, dass die Ergebnisse vereinfacht werden. Dadurch bleiben nur die wichtigsten Informationen erhalten. Des Weiteren wird dadurch erreicht, dass die 2D- oder 3D-Eingangsdaten kleiner werden. [15]

4 Vorgehensweise

4.1 Netzarchitektur

Als Basismodell dient uns das von Nvidia entworfene CNN zur Schätzung eines Steuerbefehls für ein Fahrzeug [16]. Dieses Netz nutzt fünf Faltungsschichten zur Extraktion von Merkmalen mit einer Kerngröße von (5, 5) in den ersten drei Schichten und (3, 3) in den letzten beiden. Die ersten drei Schichten verwenden außerdem ein Zero-Padding. Darauf folgen drei Dense-Schichten als Regressor.

Unsere Variante dieses Modells nutzt statt der ReLU die ELU als Aktivierungsfunktion. Diese soll den Lernprozess zusätzlich beschleunigen und die Fähigkeit des Netzes zu Generalisieren verbessern [17]. Zusätzlich haben wir eine L2 Regularisierung für die Gewichte aller Schichten und einen Dropout jeweils nach den letzten beiden Faltungsschichten hinzugefügt.

In der Vorverarbeitung der Daten wird die Größe der Bilder von (640, 480, 3) (im Format (H, W, C)) auf (120, 160, 3) reduziert und anschließend das obere Drittel des Bildes abgeschnitten, da dies meist nur Horizont und sonstige Objekte beinhaltet. Die endgültige Größe des Inputs ist dann (80, 160, 3).

Eine Darstellung unseres Netzes mitsamt Vorverarbeitung der Daten ist in Abb. 4.1 zu sehen.

4.2 Datengewinnung

Da wir uns in einer simulierten Umgebung befinden, gestaltet sich die Gewinnung von Daten mit entsprechenden Labeln als unproblematisch. Im Folgenden gehen wir auf zwei Arten möglicher Label und zwei Methoden zur Generierung der Datensätze ein.

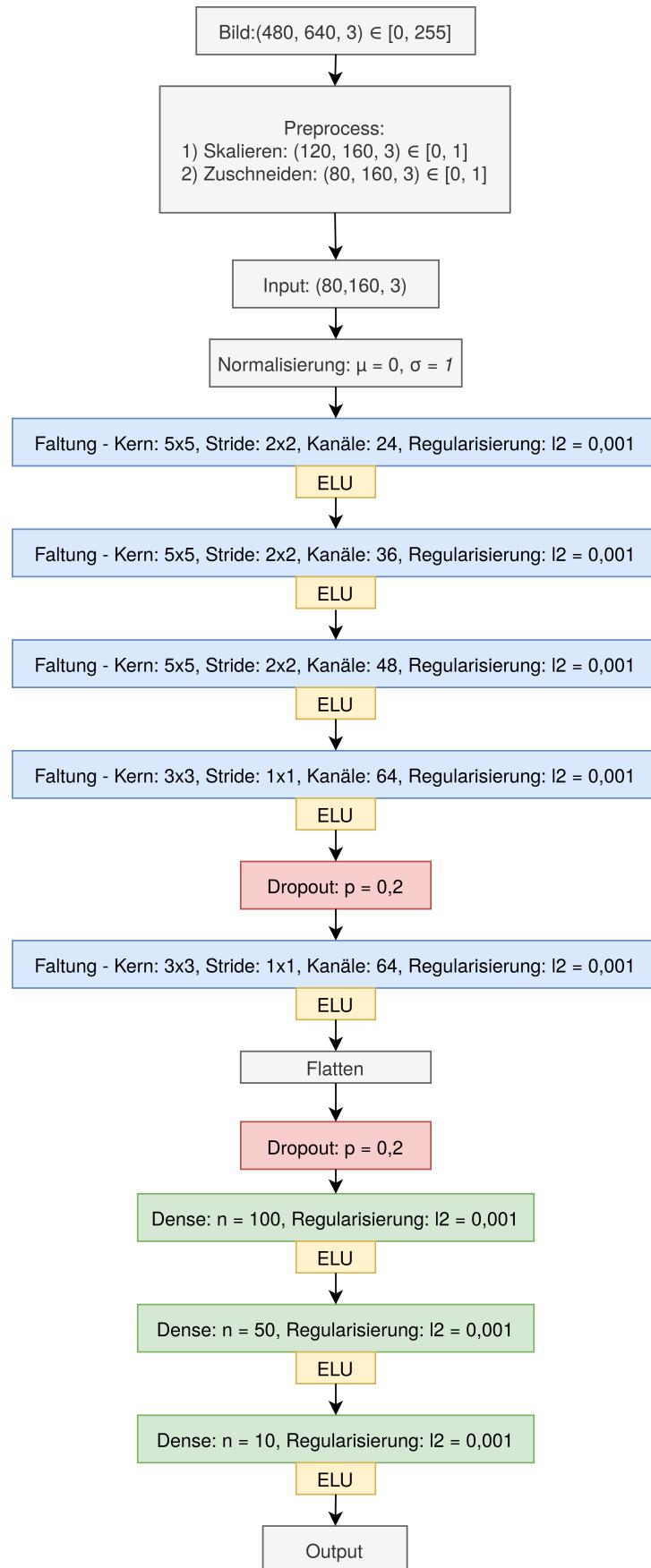


Abbildung 4.1: Unsere Variante des Nvidia-Modells

4.2.1 Label

Wir verfolgen zwei Ansätze in welchen wir das Netz auf unterschiedliche Ausgaben trainieren. Neben den beiden Werten, auf deren Berechnung im Folgenden eingegangen wird, nutzen wir noch einen dritten Wert als Label, die Art der Kachel auf welcher die Aufnahme entstanden ist.

Im ersten Ansatz ist die Aufgabe des Netzes die Schätzung der Pose des Agenten. Dazu wird vom Simulator eine Aufnahme erzeugt, die Pose des Agenten relativ zur Fahrbahn berechnet und als Label zur Aufnahme abgespeichert. Die Poseninformationen bestehen aus der Distanz zur rechten Fahrbahnmarkierung und der Differenz der Orientierungen von Agent und Tangente der Ideallinie. Mit einem einfachen PD-Regler kann dann ein Steuerbefehl für den Agenten berechnet werden, wobei die Geschwindigkeit fix gewählt wird und die Winkelgeschwindigkeit das Ergebnis des Reglers ist. Da der Distanzwert d von uns als hundertstel Kachelgröße und die Winkeldifferenz in Grad festgelegt wurde, hat eine typische Berechnung folgende Form:

Codebeispiel 4.1 Berechnung eines Steuerbefehls mit PD-Regler

```
1  d, a = environment.cheatmodul.get_lane_pose()
2  d_hat_to_center = (d - 20.5) / 100.
3  a_hat_in_rad = (a * 2 * np.pi) / 360.
4  steering = k_p * d_hat_to_center + k_d * a_hat_in_rad
5  speed = 0.2
6  observation = environment.step((speed, steering))
7  data.append((observation, (d, a)))
```

Die magische Zahl von 20.5 ist einfach die Verschiebung des Wertes vom rechten Fahrbahnrand zur Mitte der Fahrspur, so dass das Ergebnis als Fehler zum Soll (also null) interpretiert werden kann. Das Teilen durch hundert ergibt dann einen Wert in Kachelgröße, statt hundertstel Kachelgröße. Zur Vollständigkeit ist im Code noch stark vereinfacht verdeutlicht, wie Aufnahme und zugehöriges Label zustande kommen und gespeichert werden.

Das Schätzen der Pose hat den Vorteil, dass die Informationen anderweitig verwendet werden könnten, beispielsweise zur Lokalisierung des Agenten in einem Monte-Carlo Verfahren.

Im zweiten Ansatz trainieren wir das Netz direkt auf entsprechende Steuerbefehle in der Form Geschwindigkeit und Winkelgeschwindigkeit (v, ω). Dazu benutzen wir ein einfaches Expertensystem, welches mehr Informationen in die Berechnung des Befehls mitein-

bezieht, als nur die relative Pose des Agenten zum aktuellen Zeitpunkt. Dabei wird ein in einer definierten Distanz voraus liegender Punkt auf der Ideallinie gesucht. Befindet sich dieser Punkt in einer Kurve, oder sind die Orientierungen von Agent und Tangente der Fahrspur zu verschieden, wird v als die Hälfte einer Referenzgeschwindigkeit festgelegt. Bei Geraden ist v die volle Referenzgeschwindigkeit. Die Winkelgeschwindigkeit ω ist das Skalarprodukt aus dem Vektor, welcher von Roboter in Richtung voraus liegenden Punkt zeigt und dem Einheitsvektor, welcher vom Agenten aus nach rechts zeigt.

Codebeispiel 4.2 Berechnung eines Steuerbefehls mit einfachem Expertensystem

```

1 projected_angle, closest_point, curve_point = \
2     environment._get_projected_angle_difference(lookup_distance)
3 tile = environment.get_tile(curve_point)
4
5 if 'curve' in tile['kind'] or abs(projected_angle) < 0.92:
6     v *= 0.5
7
8 point_vec = curve_point - environment.cur_pos
9 point_vec /= np.linalg.norm(point_vec)
10 right_vec = np.array([math.sin(self.env.cur_angle),
11                       0, math.cos(self.env.cur_angle)])
12 omega = np.dot(right_vec, point_vec)

```

4.2.2 Observationen

Bei der Generierung der Datensätze verfolgen wir ebenfalls zwei Ansätze. Im ersten, naiven Ansatz lassen wir den Agenten selbstständig durch die Umgebung fahren, wobei für jeden Aufruf der `step()`-Methode eine Aufnahme mit entsprechendem Label gespeichert wird. Die Verteilung der Werte sollte demnach jener entsprechen, welche der Agent beim selbstständigen Fahren tatsächlich erlebt.

Im zweiten Ansatz lassen wir den Agenten nur eine gewisse Anzahl von Schritten, also `step()`-Aufrufen, fahren und rufen dann die `reset()`-Methode des Simulators auf, welche den Agenten an eine zufällige, aber valide Pose zurücksetzt. Valide bedeutet, dass die Position befahrbar und frei von Kollisionen mit anderen Objekten ist. Der Gedanke dahinter ist, dass so die Verteilung der Werte mehr „Ausnahmefälle“ beinhaltet. Denn sollte das Netz nicht im Stande sein, die Spur ähnlich gut halten zu können wie der PD-Regler oder das Expertensystem, wird es oft in Situationen geraten, auf welche es nicht trainiert wurde. Die Werteverteilung sollte so ähnlich sein zu dem, was der Agent erlebt, falls er öfters „crashen“ sollte.

4.2.3 Datensätze

Wie erläutert, ergeben sich zwei Ansätze zur Zusammensetzung des Labels und zwei Ansätze um die Verteilung der Labels zu beeinflussen. Insgesamt ergeben sich vier Datensätze. Die zwei PD-Datensätze „PD“ und „PD-Rand“ und zwei Experten-Datensätze, „Expert“ und „Expert-Rand“. Der Zusatz „Rand“ steht dabei für den höheren Anteil an Zufallsposen.

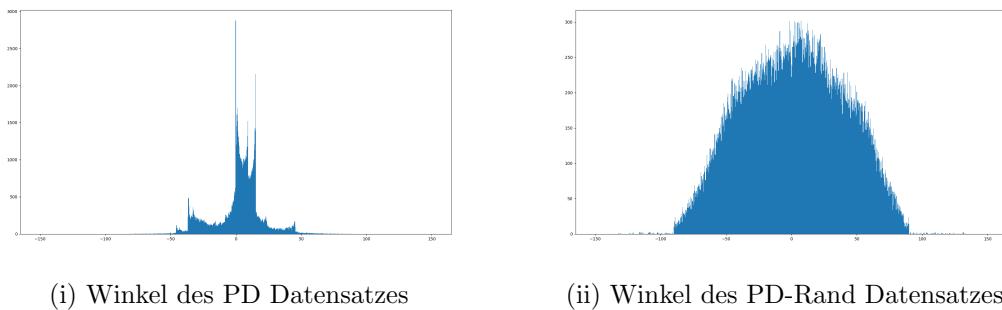


Abbildung 4.2: Verteilungen der Winkel der PD Datensätze

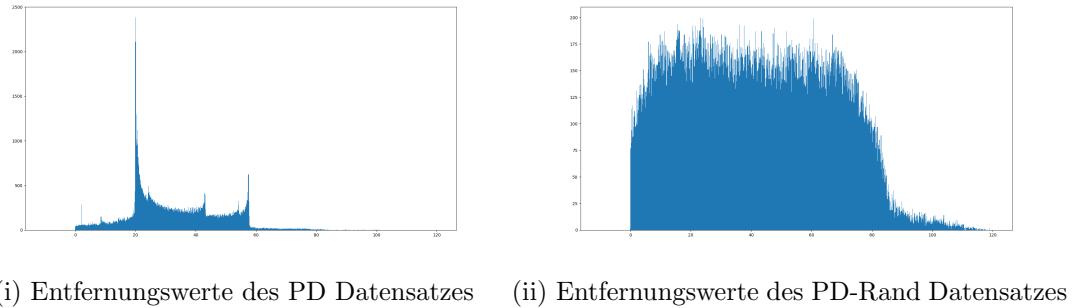
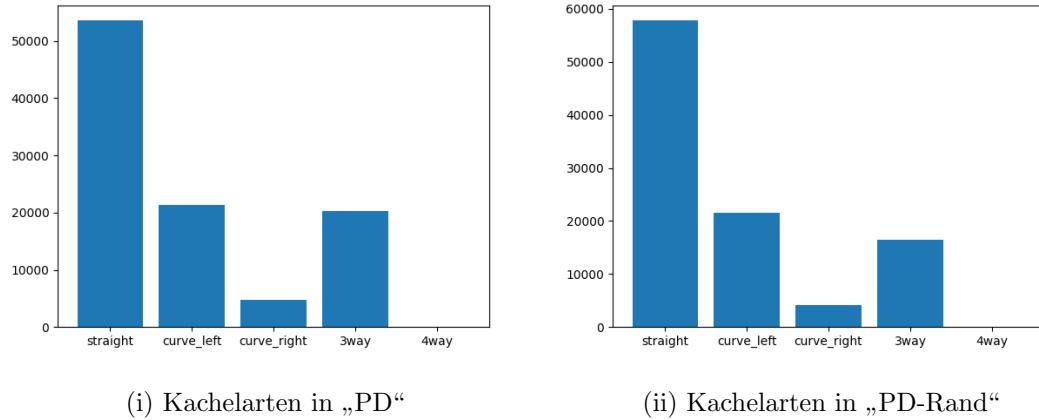


Abbildung 4.3: Verteilungen der Entfernungswerte der PD Datensätze

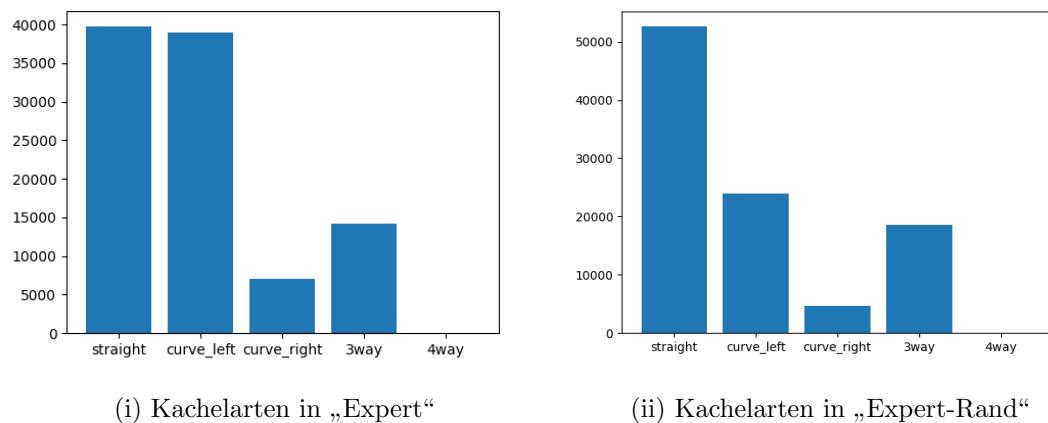
Wie in Abbildung 4.2 und 4.3 erkennbar, ist die Verteilung von sowohl Entfernungswerten, als auch Winkeldifferenzen im „PD-Rand“ Datensatz wesentlich gleichmäßiger als im „PD“-Datensatz.



(i) Kachelarten in „PD“

(ii) Kachelarten in „PD-Rand“

Abbildung 4.4: Verteilungen der Kachelarten in „PD“ und „PD-Rand“



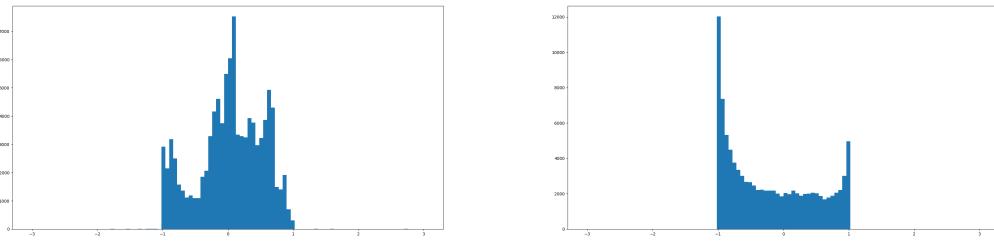
(i) Kachelarten in „Expert“

(ii) Kachelarten in „Expert-Rand“

Abbildung 4.5: Verteilungen der Kachelarten in „Expert“ und „Expert-Rand“

Auf die Verteilung der Kachelarten in den PD-Datensätzen, auf welchen die Aufnahmen entstanden sind, hat der höhere Anteil der Zufallsposen keinen Einfluss (siehe Abb. 4.4). Im Gegensatz dazu hat der höhere Anteil an Zufallsposen in den Expertendatensätzen Auswirkungen auf die Verteilung der Kachelarten. So ist der Anteil der Linkskurven im Datensatz deutlich gestiegen (siehe Abb. 4.5).

Vier-Wege-Kreuzungen waren auf der von uns verwendeten Umgebungskarte nicht vorhanden.



(i) Winkelgeschwindigkeiten des Expert Daten- (ii) Winkelgeschwindigkeiten des Expert-Rand
satzes Datensatzes

Abbildung 4.6: Verteilungen der Winkelgeschwindigkeiten der Expert Datensätze

In den Expertendatensätzen 4.6 lässt sich ebenfalls ein deutlicher Unterschied in der Verteilung der berechneten Winkelgeschwindigkeiten des Steuerbefehls feststellen. In Abbildung 4.6ii ist eine Menge an Maximalwerten von -1 und 1 erkennbar (der Wertebereich durch das Skalarprodukt, kann zum Beispiel mit Faktor π multipliziert werden). Auf die berechneten Geschwindigkeiten der Expertendatensätze muss nicht eingegangen werden, da zum einen aufgrund unseres einfachen Expertensystems die Verteilung nur aus zwei Werten besteht und zum Anderen sich diese Verteilung durch die Ansätze nicht wesentlich unterscheidet.

Datensatz	Größe
PD	100000
PD-Rand	100000
Expert	99943
Expert-Rand	99812

Abbildung 4.7: Größe der erzeugten Datensätze

4.3 Lernprozess

Für einen erfolgreichen Lernprozess ist zunächst die Wahl der Hyperparameter festzulegen, um eine stabile Konvergenz zu ermöglichen. Anschließend kann diese Konfiguration auf Kombinationen von Datensätzen angewandt und verglichen werden.

4.3.1 Hyperparameter

Das Feintuning des Modells und die Wahl der Hyperparameter wurden in einem langwierigen Prozess experimentell erarbeitet.

Lerngeschwindigkeit	0.0002
Batch Größe	32
Optimizer	Adam
Anzahl Epochen	50
Verlustfunktion	MSE

Wir setzen den Adam Optimizer ein, da dieser durch adaptive Lernraten oft einen schnelleren Lernprozess ermöglicht und generell in der Literatur oft zum Einsatz kommt. Wir parametrisieren Adam mit einer initialen Lernrate von 0.0002. Diese ergab in Kombination mit einer Batchgröße von 32 in den meisten Fällen eine gute Konvergenz.

Die Größe der Batch von 32 soll einen Kompromiss bilden zwischen Geschwindigkeit des Prozesses und der Fähigkeit des Netzes zu Generalisieren. Eine Batchgröße über 32 hat sich bereits als unvorteilhaft erwiesen [18].

Als Fehlerfunktion wären prinzipiell sowohl MSE als auch MAE vorstellbar. Eine L2-Fehlerfunktion hat den Vorteil, dass Ausreißer in den Daten durch das Quadrat in der Funktion härter bestraft werden. Unsere Daten weisen allerdings nur wenige Ausreißer auf, so dass sich in Experimenten die MAE nicht deutlich von der MSE unterschieden hat. Da MSE in der Literatur häufiger zum Einsatz kommt, blieben wir mit unserer Wahl bei ihr.

Den Prozess ließen wir mit einer Dauer von 50 Epochen laufen. Häufig ist schon nach wenigen Epochen erkennbar, ob die Kombination an gewählten Parametern eine gute Konvergenz ergibt. Aber sollte die Konfiguration konvergieren, geben 50 Durchläufe genug Zeit ein Minimum zu finden.

4.3.2 Anwendung

Wir haben vier Datensätze und kombinieren die Datensätze der beiden Ansätzen zur Erzeugung (ein mal durch einfaches Zusehen und ein mal durch häufiges Zurücksetzen), so dass sich insgesamt sechs ergeben („PD“, „PD-Rand“, „PD + PD-Rand“, „Expert“, „Expert-Rand“, „Expert + Expert-Rand“). Natürlich muss beachtet werden, dass die beiden kombinierten Datensätze etwa doppelt so groß sind, wie die übrigen. Wir wählen eine Aufteilung der Datensätze von 70 % und 30% für Trainings- und Testdaten. Eine Validierung wird anschließend durch eine Integration des Netzes in den Simulator durchgeführt.

Außerdem variieren wir den Output unseres Netzes. Beim Inferieren der relativen Pose testen wir, ob sich Unterschiede in der Performance des Netzes ergeben, je nach dem

ob ein zwei- oder eindimensionaler Zustandsvektor geschätzt werden soll. Wir trainieren dabei ein mal nur auf den Distanzwert zur Fahrbahnbegrenzung und ein mal auf sowohl Distanz, als auch Winkeldifferenz.

Das Gleiche testen wir ebenfalls an unserem Experten-Ansatz. Hier entfernen wir die Geschwindigkeit als Ziel beim Training und verwenden nur die wesentlich wichtigere Winkelgeschwindigkeit.

5 Ergebnisse

Zunächst testen wir, wie gut welche Art von Output während Trainings- und Testphase abschneidet. Dazu vergleichen wir die Performance unter Variation des Outputs mit den jeweils drei entsprechenden Datensätzen.

Anschließend integrieren wir das Netz in den DuckieTown-Simulator und bewerten die tatsächliche Performance in einer Validierungsphase.

5.1 Training und Test

Wie zu erwarten ist der Fehler des Netzes stark abhängig von der Verteilung der Trainingsdaten. So ergab sich in allen Fällen ein höherer Fehler, je größer der Anteil an Zufallsposen im Datensatz.

5.1.1 2D-Pose

Beim Training auf eine 2D-Pose fällt auf, dass das Schätzen des Winkels θ eine größere Herausforderung darstellt, als das Schätzen der Distanz d . Abbildung 5.2 zeigt, dass bei dem Training mit Daten mit hohem Anteil an Zufallsposen (Datensatz „PD-Rand“ in den Farben Grün und Lila) das Netz etwa ab der zehnten Epoche überstimmt ist. Im Vergleich mit den absoluten Fehlern der einzelnen Dimensionen in Abb. 5.3 und 5.4, ist festzustellen, dass die Überstimmung vor allem auf Schwierigkeiten mit dem Schätzen des Winkels θ zurückzuführen ist.

Beim Schätzen der Distanz d ist selbst mit dem „PD-Rand“ Datensatz in Epoche 50 die Überstimmung des Netzes minimal, wie in Abbildung 5.3 zu sehen ist.

Datensatz	MSE	MAE d	MAE θ
PD	8.164	1.404	1.784
PD-Rand	274.2	8.312	12.61
PD + PD-Rand	143.8	5.353	8.036

Abbildung 5.1: Performance der 2D-Posenschätzung auf Testdaten

5 Ergebnisse

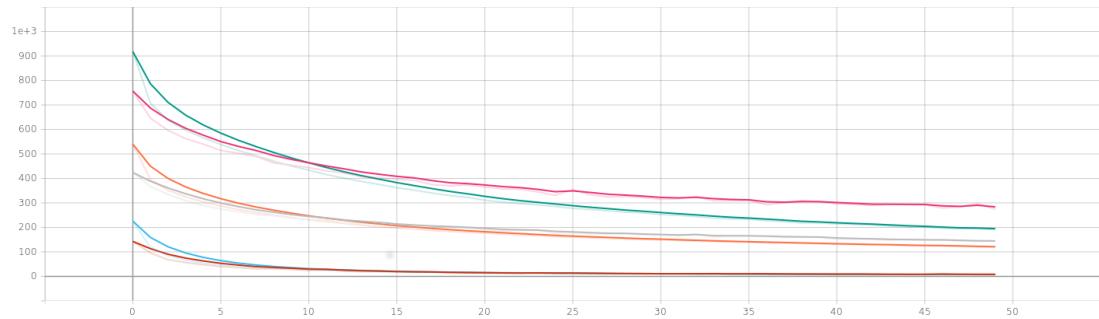


Abbildung 5.2: MSE mit 2D-Poseabschätzung

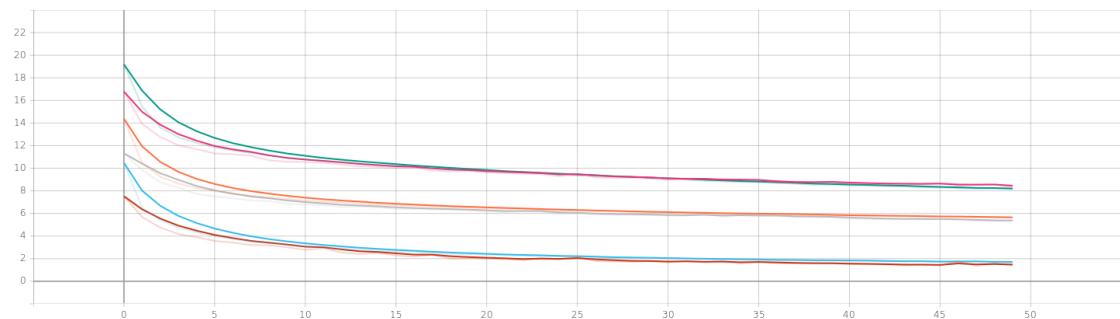


Abbildung 5.3: MAE der Distanz mit 2D-Poseabschätzung

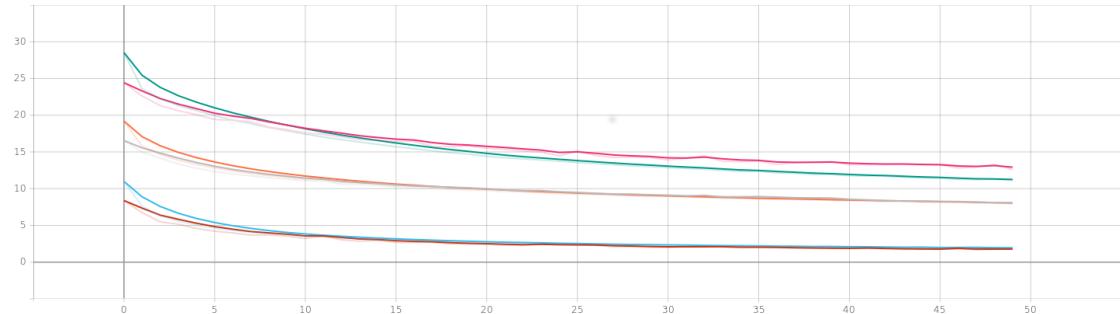


Abbildung 5.4: MAE des Winkels mit 2D-Poseabschätzung

5.1.2 1D-Pose

Das Entfernen des Winkels aus dem Zustandsvektor bringt eine leichte Verbesserung der Performance auf allen Datensätzen mit sich (siehe Abb. 5.5 und 5.1).

Datensatz	MSE	MAE d
PD	3.423	1.104
PD-Rand	115.4	6.73
PD + PD-Rand	60.0	4.364

Abbildung 5.5: Performance der 1D-Poseabschätzung auf Testdaten

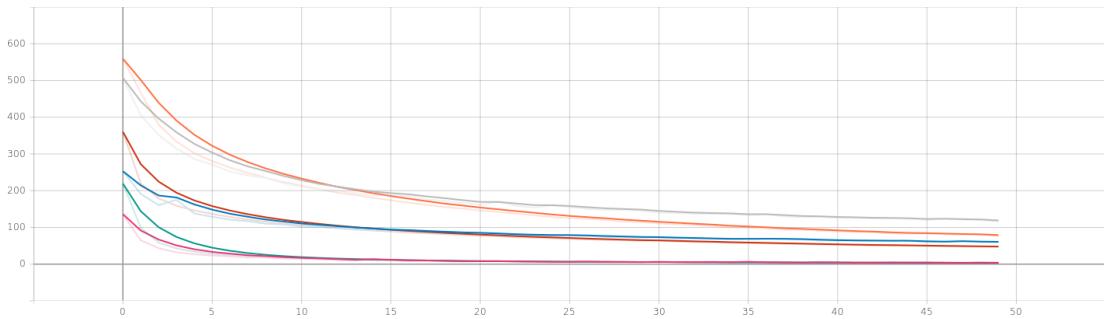


Abbildung 5.6: MSE der Distanz mit 1D-Posenschätzung

Allerdings führt die Reduktion auf das Schätzen des Distanzwertes allein zu einer stärkeren Überstimmung des Netzes. In Abbildung 5.6 ist erkennbar, dass sowohl bei dem Training mit dem „PD-Rand“ (hier in Orange für die Trainingsdaten und Grau für die Testdaten), als auch mit dem „PD + PD-Rand“ Datensatz (Rot für Training und Blau für Test), das Netz spätestens ab Epoche 15 überstimmt ist.

5.1.3 Steuerbefehl durch Expertensystem

Wie zuvor zeigt sich auch hier, dass das Training mit hohem Anteil an Zufallsposen in den Daten zu einer Überstimmung des Netzes führt. So bilden sich deutliche Lücken zwischen der Performance auf Trainings- und Testdaten auf den Datensätzen „Expert-Rand“ (Test Grau und Training Orange) und „Expert + Expert-Rand“ (Test Blau und Training Rot) in Abbildung 5.8.

Datensatz	MSE	MAE ω
Expert	0.00536	0.04214
Expert-Rand	0.11	0.205
Expert + Expert-Rand	0.057	0.1322

Abbildung 5.7: Performance der Schätzung eines Steuerbefehls auf Testdaten

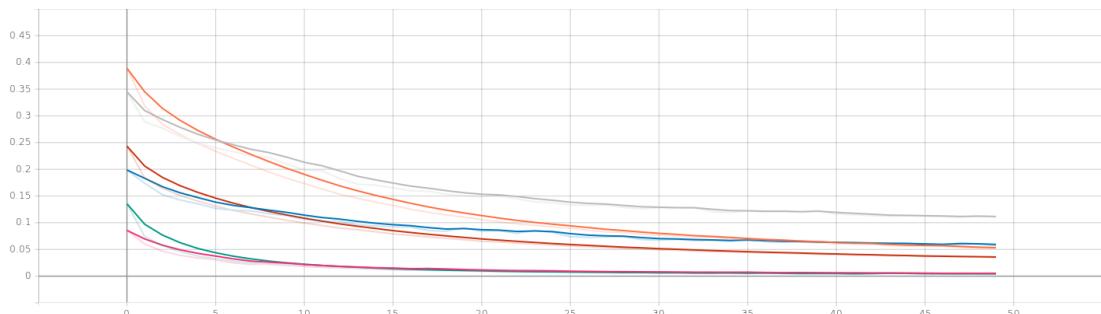


Abbildung 5.8: MSE der Winkelgeschwindigkeit mit Expertenbefehl

5.2 Validierung

Die Validierung besteht aus der Integration des Netzes in den Simulator, wobei dieses selbst die Kontrolle übernimmt. Die Steuerung erfolgt dabei in Abhängigkeit von dem Output des Netzes. Im Falle des 2D-Posen-Netzes über einen einfachen PD-Regler mit d als Fehler und θ als Ableitung des Fehlers (siehe Codebeispiel 2.2). Das 1D-Posen-Netz hält den Distanzwert über einen vollständigen und sorgfältig eingestellten PID-Regler. Das Experten-Netz steuert den Agenten direkt über den inferierten Steuerbefehl.

Im Gegensatz zur Trainings- und Testphase ist bei der Integration in den Simulator die Verteilung der Werte mit denen das Netz konfrontiert wird abhängig davon, wie gut dieses die Spur halten kann. Dies ist wiederum abhängig davon, wie gut die letzten Werte geschätzt worden sind.

Wir lassen die Netze jeweils 100000 `step()`-Aufrufe des Simulators fahren, was bei einer Bildrate von 30 Bildern pro Sekunde einer Fahrtdauer von etwa einer Stunde entspricht.

5.2.1 2D-Pose

Im ersten Fall des 2D-Posen-Netzes schnitt das Netz, welches mit dem „PD-Rand“-Datensatz und damit dem höchsten Anteil an Zufallsposen trainiert wurde, in Sachen Crashes pro Schritte am besten ab (siehe Tabelle 5.1). Auffällig ist, dass das Training mit „PD-Concat“ zwar zu einen besseren durchschnittlichen Fehler führte, was auch mit dem doppelt so großen Umfang des Datensatzes zu erklären ist, jedoch nicht zu der geringsten Anzahl an Crashes.

	Kacheltyp	MAE d, θ	n	Crashes
PD	Alle	12.59, 20.41	100000	790
	Gerade	9.76, 13.79	53433	
	Linkskurve	16.70, 35.28	20311	
	Rechtskurve	23.78, 37.95	4728	
	3-Wege	13.27, 18.95	21528	
PD-Rand	Alle	11.63, 17.19	100000	485
	Gerade	9.67, 13.08	55325	
	Linkskurve	14.38, 26.80	18513	
	Rechtskurve	16.44, 31.53	3796	
	3-Wege	13.36, 16.97	22366	
PD-Concat	Alle	10.62, 16.37	100000	592
	Gerade	7.67, 11.04	49792	
	Linkskurve	13.73, 23.53	24695	
	Rechtskurve	17.59, 29.68	5163	
	3-Wege	12.28, 17.35	20350	

Tabelle 5.1: Validierung des 2D-Posen-Netzes

In der Verteilung der Werte, mit welchen das Netz während der Integration konfrontiert

wurde, ist das bessere Fahrverhalten des mit dem „PD-Rand“-Datensatz trainierten Netzes ebenfalls erkennbar (siehe Abb. 5.9). So liegt der Schwerpunkt der Entfernungswerte eher nahe 20, was der Mitte der rechten Fahrspur entspricht. Auch die Winkel streuen eher um 0, was eine gute Orientierung während des Fahrens anzeigt.

Die anderen beiden Varianten befanden sich während der Fahrt eher auf der gegenüberliegenden Fahrspur (Schwerpunkt der Distanzwerte um 60) und neigten zu einer linkslastigen Orientierung (Schwerpunkt der Winkel unter 0).

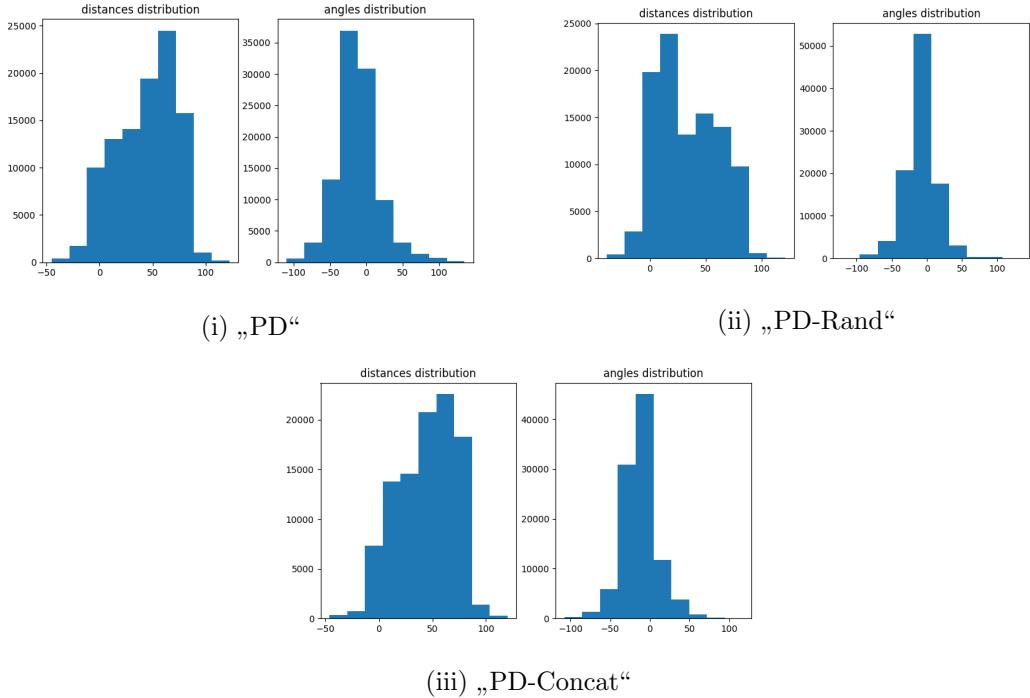


Abbildung 5.9: Verteilungen der erlebten Werte während der Validierung des 2D-Netzes

5.2.2 1D-Pose

Bei den 1D-Posen-Netzen wurden die besten Ergebnisse mit dem „PD“-Datensatz erzielt (siehe Tabelle 5.2). Sowohl der durchschnittliche Fehler während der Validierung, als auch die Anzahl der Crashes waren in dieser Variante besonders niedrig.

Das Training mit kombiniertem Datensatz ergab hier die schlechteste Performance, obwohl sich wieder ein geringerer durchschnittlicher Fehler ergab, als bei dem Training mit vielen Zufallsposen im Datensatz.

	Kacheltyp	MAE d	n	Crashes
PD	Alle	5.32	100000	1
	Gerade	3.09	45990	
	Linkskurve	9.77	34287	
	Rechtskurve	7.85	4403	
	3-Wege	1.34	15320	
PD-Rand	Alle	14.44	100000	27
	Gerade	9.73	7974	
	Linkskurve	13.01	1324	
	Rechtskurve	14.83	89228	
	3-Wege	17.22	1474	
PD-Concat	Alle	8.37	100000	49
	Gerade	6.36	80467	
	Linkskurve	16.04	9792	
	Rechtskurve	23.76	4276	
	3-Wege	12.19	5465	

Tabelle 5.2: Validierung des 1D-Posen-Netzes

Bestätigt wird das Ergebnis wieder durch die Verteilung der Posenwerte während der Validierung (siehe Abb. 5.10). Das mit dem „PD“-Datensatz trainierte Netz war am besten im Stande den Abstand zur Fahrbahnbegrenzung von etwa 20-25 halten zu können. Die anderen beiden Varianten waren mit Peaks um 40 und 60 wieder eher in der Mitte der Straße bis auf der linken Fahrspur unterwegs.

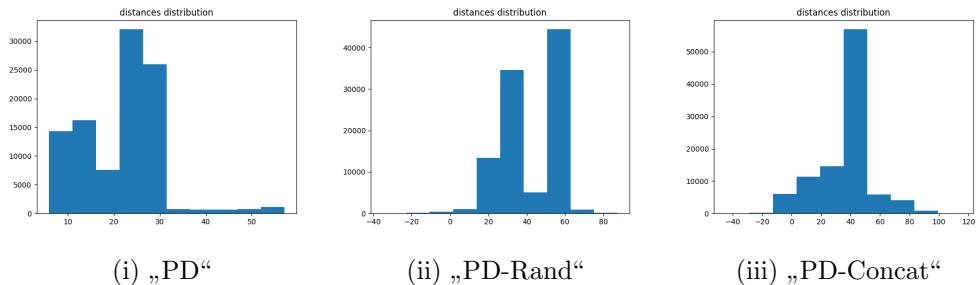


Abbildung 5.10: Verteilungen der erlebten Werte während der Validierung des 1D-Netzes

5.2.3 Steuerbefehl durch Expertensystem

Die auf Steuerbefehle trainierten Netze lieferten bessere Ergebnisse je geringer der Anteil an Zufallsposen im Datensatz (siehe Tabelle 5.3). Dies gilt sowohl für die Anzahl an Crashes, als auch den durchschnittlichen Fehler. Auch hier ist das beste Fahrverhalten wieder an der Verteilung der erlebten Werte erkennbar (siehe Abb. 5.11). So liegt das Mittel der Werte näher an 0, je geringer der Anteil an Zufallsposen im Datensatz.

	Kacheltyp	MAE ω	n	Crashes
Expert	Alle	0.096	100000	2
	Gerade	0.064	49249	
	Linkskurve	0.162	31399	
	Rechtskurve	0.126	3002	
	3-Wege	0.061	16350	
Expert-Rand	Alle	0.506	100000	311
	Gerade	0.486	48791	
	Linkskurve	0.573	28062	
	Rechtskurve	0.715	1657	
	3-Wege	0.446	21490	
Expert-Concat	Alle	0.150	100000	34
	Gerade	0.117	47454	
	Linkskurve	0.212	31424	
	Rechtskurve	0.199	3888	
	3-Wege	0.117	17234	

Tabelle 5.3: Validierung des Steuerbefehl-Netzes

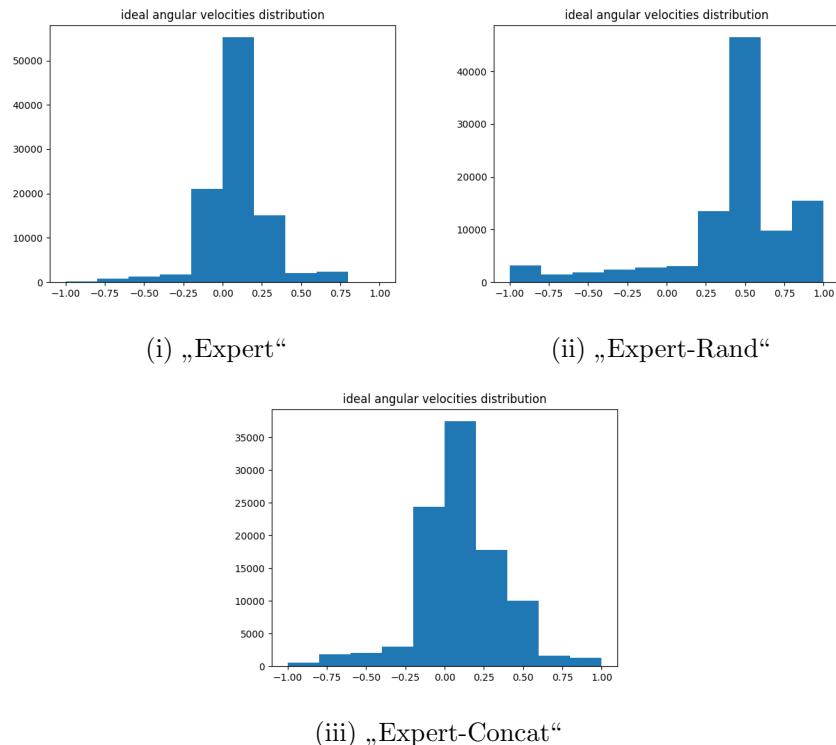


Abbildung 5.11: Verteilungen der erlebten Werte während der Validierung des Experten-Netzes

6 Fazit

In dieser Arbeit wird beschrieben, wie mit Hilfe eines tiefen neuronalen Netzes die relative Pose eines Agenten in einer simulierten Umgebung inferiert werden kann. Mit dem besonderen Fokus auf das Thema Liniенverfolgung, entwickeln wir dabei mehrere Ansätze. Zu Gunsten von stabilerem Fahrverhalten testen wir außerdem die Performance von direkter Inferenz eines Steuerbefehls für einen DuckieTown Agenten.

6.1 Rückblick

Im Folgenden reflektieren wir über die Ergebnisse der verfolgten Ansätze.

6.1.1 2D-Pose

Das Schätzen einer zweidimensionalen Pose erwies sich als am Schwierigsten.

Während des Trainings mit dem „PD“-Rand Datensatz konvergierte der Fehler in d etwas besser als θ . Bildmerkmale für das Schätzen der einzelnen Werte scheinen damit verschieden und durch den erhöhten Anteil an Zufallsposen auch unterschiedlich stark vertreten zu sein.

Obwohl die Metriken während dem Training mit den Daten mit dem geringsten Anteil an Zufallsposen am besten wirkten, zeigte die Integration des Netzes ein sehr schlechtes Fahrverhalten mit 790 Unglücken in 100000 Schritten. Am besten schnitt das Netz mit dem größten Anteil an Zufallsposen im Datensatz ab, obwohl dies den größten Fehler zeigte. Das Netz, welches mit dem kombinierten Datensatz trainiert wurde, lag in der Mitte.

Eine mögliche Erklärung dafür ist die eher geringe Reaktionsgeschwindigkeit und die Tendenz zum Überschwingen des einfachen PD-Reglers. Durch diese kommt der Agent häufig in Situationen, in welchen er sich eher am Fahrbahnrand und abgeneigt von der Orientierung der Straße befindet. Während das Fahren mit Grundwahrheit mit diesen Überschwingern zurecht kam, können Fehler in der Schätzung hier schnell zu einem Überfahren der Fahrbahnmarkierung führen. Schon geringe Fehler in den Schätzungen führten hier möglicherweise zu einer Abweichung der während der Integration erlebten Verteilung der Input-Daten von der Verteilung der Trainingsdaten. Das Netz konnte also von dem erhöhten Anteil an Zufallsposen im Datensatz profitieren, da dieser eher der tatsächlich in der Integration erlebten Situation ähnelte. Zusätzlich entsteht durch die lineare Kombination der beiden geschätzten Dimensionen durch $\omega = k_p d + k_d \theta$ ein höherer Fehler von $\delta\omega = \sqrt{(k_p \delta d)^2 + (k_d \delta \theta)^2}$.

Nach dieser Erklärung müsste das Training mit kombiniertem Datensatz eigentlich das zweitbeste Ergebnis liefern, zumal dieses sogar den geringsten durchschnittlichen Fehler aufweist. Wir halten die Erklärung dennoch für plausibel, da die Werte der Ansätze

mit kombiniertem Datensatz und Datensatz mit hohem Anteil an Zufallsposen nahe beieinander liegen und wir hier lediglich eine Stichprobe besitzen.

6.1.2 1D-Pose

Das Entfernen der Winkeldifferenz θ aus dem zu inferierenden Zustandsvektor verbesserte den Fehler in d mit den Datensätzen „PD-Rand“ und „PD-Concat“ um etwa die Hälfte (siehe Abb. 5.1 und 5.5). Dies könnte bedeuten, dass das Minimieren der Fehler in den einzelnen Dimensionen des 2D Zustandes in Konkurrenz zueinander steht.

Während der Integration in den Simulator lieferte der 1D Ansatz sehr gute Ergebnisse. Hier führte das Training mit „PD“ zum besten Ergebnis an Crashes und MAE. Das Training mit kombiniertem Datensatz verunglückte dagegen am häufigsten, obwohl es einen besseren mittleren Fehler hervorbrachte als das Training mit „PD-Rand“.

Eine gute Reaktionsgeschwindigkeit des PID-Reglers und das Wegfallen der Fehlerfortpflanzung sorgten hier dafür, dass die Verteilung der während der Validierung erlebten Werte der Verteilung der Trainings- und Testdaten ähnelte. Das schlechte Abschneiden des mit kombiniertem Datensatz trainierten Netzes kann mit einem hohen Fehler in Kurvenabschnitten erklärt werden, wo das Halten der Spur am schwierigsten ist.

6.1.3 Steuerbefehl durch Expertensystem

Bei dem Training auf Steuerbefehle führte der erhöhte Anteil an Zufallsposen im Datensatz zu einer starken Überstimmung des Netzes (siehe 5.8). Auch die Performance während der Validierung war hier mit Abstand am schlechtesten (siehe 5.3). Die Kombination der Datensätze verschlechterte das Ergebnis ebenfalls. Dagegen konnte durch das Training ohne zusätzliche Zufallsposen in den Daten ein sehr gutes Ergebnis von nur zwei Crashes erzielt werden.

Ohne den zusätzlichen Aufwand eines PID-Reglers kann hier die Spur so gut gehalten werden, dass ein Training mit besonderem Fokus auf Ausnahmefälle (also Zufallsposen) keine Verbesserung, sondern eine Verschlechterung mit sich bringt. Dass das Training mit kombiniertem Datensatz ebenfalls schlechter Abschneidet, zeigt, dass die Menge an Zufallsposen den Lernprozess behindern und das Netz von den eigentlich zu erlernenden Merkmalen ablenken.

6.2 Verbesserungen

Abschließend diskutieren wir mögliche Verbesserungen der verfolgten Ansätze.

6.2.1 2D-Pose

Die anscheinend konkurrierende Minimierung der Fehler in den einzelnen Dimensionen des Zustandsvektors lässt Zweifel an der Eignung der Netzarchitektur für diese Art

von Problem aufkommen. Andere Ansätze zur Schätzung einer Pose verwenden meist komplexere Architekturen.

Eine mögliche Verbesserung wäre mehrere parallele Regressoren in Form von vollständig verbundenen Schichten für jede Dimension des zu schätzenden Vektors zu verwenden, anstatt nur einem, wie in anderen Arbeiten zu sehen ist (siehe [19]). Eine gute Fähigkeit in beiden Dimensionen zu generalisieren könnte so ein stabiles Fahrverhalten ermöglichen.

6.2.2 1D-Pose

Das stabile Fahrverhalten des 1D-Posen-Netzes mit PID-Regler würde es erlauben weiter darauf aufzubauen. Wie bereits in der Aufgabenstellung (siehe 1.1) erläutert, könnte in Kombination mit einer Umgebungskarte, Odometrie und einem Monte-Carlo-Verfahren das Problem der globalen Lokalisierung angegangen werden.

Eine weitere Reduktion des Fehlers in der Schätzung wäre mit entsprechendem Datensatz und einer Anpassung der Hyperparameter ebenfalls denkbar.

6.2.3 Steuerbefehl durch Expertensystem

Auch hier wäre eine Weiterentwicklung zum Zweck der globalen Lokalisierung denkbar. Das Erkennen der durchfahrenden Kachelart wäre langsamer als im Falle des 1D-Posen Ansatzes, da keine Distanzwerte vorhanden sind, mit Hilfe derer die Änderung der globalen Orientierung verrechnet werden kann (zur Unterscheidung von Kurvenabschnitten und dem Fahren von Schlangenlinien).

Weiterhin kann das verwendete Expertensystem zur Generierung der Steuerbefehle weiter ausgebaut werden. Die Logik zur Berechnung der Geschwindigkeit v war in unserem Fall einfach eine Halbierung der Referenz im Falle einer Kurve. Zu Gunsten der Winkelgeschwindigkeit wurde daher der Wert aus dem Training entfernt. Ein intelligenterer Ansatz zur Berechnung, womöglich in Zusammenhang mit der vorher bestimmten Winkelgeschwindigkeit, könnte dazu führen, dass eine schnellere Durchschnittsgeschwindigkeit erreicht werden kann. In unserem Ansatz wählten wir dagegen eine feste und eher langsamere Geschwindigkeit.

Abbildungsverzeichnis

1.1	DuckieTown	1
2.1	Darstellung einer beispielhaften DuckieTown-Umgebung	4
2.2	Darstellung eines DuckieBots	6
3.1	Übersicht Künstliche Intelligenz	8
3.2	Darstellung eines beispielhaften künstlichen neuronalen Netzes (vereinfacht)	9
3.3	Darstellung der Performance von Deep Learning Algorithmen im Vergleich zu älteren Lernalgorithmen	10
3.4	Darstellung der verschiedenen Lernverfahren	11
3.5	Lineare Klassifikation versus nichtlineare Klassifikation	18
3.6	Darstellung eines beispielhaften CNN	18
4.1	Unsere Variante des Nvidia-Modells	20
4.2	Verteilungen der Winkel der PD Datensätze	23
4.3	Verteilungen der Entfernungswerte der PD Datensätze	23
4.4	Verteilungen der Kachelarten in „PD“ und „PD-Rand“	24
4.5	Verteilungen der Kachelarten in „Expert“ und „Expert-Rand“	24
4.6	Verteilungen der Winkelgeschwindigkeiten der Expert Datensätze	25
4.7	Größe der erzeugten Datensätze	25
5.1	Performance der 2D-Posenschätzung auf Testdaten	27
5.2	MSE mit 2D-Posenschätzung	28
5.3	MAE der Distanz mit 2D-Posenschätzung	28
5.4	MAE des Winkels mit 2D-Posenschätzung	28
5.5	Performance der 1D-Posenschätzung auf Testdaten	28
5.6	MSE der Distanz mit 1D-Posenschätzung	29
5.7	Performance der Schätzung eines Steuerbefehls auf Testdaten	29
5.8	MSE der Winkelgeschwindigkeit mit Expertenbefehl	29
5.9	Verteilungen der erlebten Werte während der Validierung des 2D-Netzes .	31
5.10	Verteilungen der erlebten Werte während der Validierung des 1D-Netzes .	32
5.11	Verteilungen der erlebten Werte während der Validierung des Experten-Netzes	33

Codeauflistung

2.1	Beispieldefinition einer DuckieTown-Umgebung	6
2.2	Bedienung einer DuckieTown-Umgebung	7
4.1	Berechnung eines Steuerbefehls mit PD-Regler	21
4.2	Berechnung eines Steuerbefehls mit einfachem Expertensystem	22

Literaturverzeichnis

- [1] Duckietown Foundation. The duckietown foundation. <https://www.duckietown.org/about/duckietown-foundation>. Abgerufen: 26.08.2020.
- [2] OpenAI. Gym. <https://gym.openai.com/>. Abgerufen: 17.10.2020.
- [3] Maxime Chevalier-Boisvert, Florian Golemo, Yanjun Cao, Bhairav Mehta, and Liam Paull. Duckietown environments for openai gym. <https://github.com/duckietown/gym-duckietown>, 2018.
- [4] Laurenz Wuttke. Deep learning: Einführung, Beispiele & Frameworks. <https://datasolut.com/was-ist-deep-learning/>. Abgerufen: 02.10.2020.
- [5] Laurenz Wuttke. Machine learning: Definition, Algorithmen, Methoden und Beispiele. <https://datasolut.com/was-ist-machine-learning/>. Abgerufen: 08.10.2020.
- [6] Mathias Sauermann. Machine Learning: 3 KI-Lernverfahren auf einen Blick. <https://der-onliner.blogspot.com/2019/07/machinelles-lernen-lernverfahren.html>, 2019. Abgerufen: 08.10.2020.
- [7] Laurenz Wuttke. Training-, Validierung- und Testdatensatz. <https://datasolut.com/wiki/trainingsdaten-und-testdaten-machine-learning/>, 2020. Abgerufen: 30.09.2020.
- [8] jaai.de. Künstliche neuronale Netze – Aufbau & Funktionsweise. <https://jaai.de/kuenstliche-neuronale-netze-aufbau-funktion-291/>. Abgerufen: 08.10.2020.
- [9] divis.io. KI leicht erklärt – Teil 4: Die Grundlagen des Machine Learning. <https://divis.io/2019/04/ki-leicht-erklaert-teil-4-die-grundlagen-des-machine-learning/>, 2019. Abgerufen: 09.10.2020.
- [10] wikipedia.org. Hyperparameteroptimierung. <https://de.wikipedia.org/wiki/Hyperparameteroptimierung>. Abgerufen: 09.10.2020.
- [11] ai united.de. Aktivierungsfunktionen, ihre Arten und Verwendungsmöglichkeiten. <http://www.ai-united.de/aktivierungsfunktionen-ihre-arten-und-verwendungsmoeglichkeiten/>. Abgerufen: 08.10.2020.
- [12] rocketloop.de. Was sind künstliche neuronale Netze? <https://rocketloop.de/kuenstliche-neuronale-netze/>. Abgerufen: 09.10.2020.
- [13] Lucas Plagwitz. Numerische Optimierung für Deep Learning Algorithmen. https://www.uni-muenster.de/AMM/num/Vorlesungen/Seminar_Wirth-Master_WS18_b/handouts/Plagwitz.pdf. Abgerufen: 09.10.2020.

- [14] Oliver Gableske. MULTILAYER-PERZEPTRON. <https://www.informatik.uni-ulm.de/ni/Lehre/WS04/ProSemNN/pdf/MLP.pdf>. Abgerufen: 20.10.2020.
- [15] Laurenz Wuttke. Künstliche Neuronale Netzwerke: Definition, Einführung, Arten und Funktion. <https://datasolut.com/neuronale-netzwerke-einfuehrung/>. Abgerufen: 09.10.2020.
- [16] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to End Learning for Self-Driving Cars. *arXiv e-prints*, page arXiv:1604.07316, April 2016.
- [17] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus), 2016.
- [18] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima, 2017.
- [19] Yoli Shavit and Ron Ferens. Introduction to camera pose estimation with deep learning, 2019.