UNIVERSITÄT ZU LÜBECK

Erlernen von Bewegungsmodellen für lokale Pfadplanungsstrategien

Learning Motion Models for Local Path Planning Strategies

Bachelorarbeit

verfasst am
Institut für Robotik und Kognitive Systeme

im Rahmen des Studiengangs
Robotik und Autonome Systeme
der Universität zu Lübeck

vorgelegt von
Leander Busch

ausgegeben und betreut von
Prof. Dr. Elmar Rückert

mit Unterstützung von
Nils Rottmann

Lübeck, den 15. Februar 2021

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

_____

Leander Busch

Zusammenfassung

Der *Segway Loomo* ist ein selbstbalancierender Segwayroboter, der durch eine interne Regelung ständig ausbalanciert wird. Für diesen Roboter wurde im Vorfeld eine lokale Pfadplanungsstrategie entwickelt. Für die lokale Pfadplanung wird ein Bewegungsmodell des Roboters benötigt, um die Auswirkung von Geschwindigkeitsbefehlen an den Roboter auf die Stellung des Roboters zu bestimmen. Bei dem implementierten lokalen Pfadplaner wird ein einfaches Bewegungsmodell des Roboters benutzt, das den Einfluss der internen Regelung des Segwayroboters auf dessen Bewegung nicht modelliert. In dieser Arbeit wurde untersucht, ob ein genaueres Bewegungsmodell für den *Segway Loomo* Roboter erlernt werden kann, um die lokale Pfadplanung für diesen Roboter zu verbessern. Für das Erlernen des Bewegungsmodell wurden künstliche neuronale Netze eingesetzt. Hierfür wurden verschiedene Architekturen von Feedforward-Netzen getestet. Die neuronalen Netze wurden anhand aufgenommener Bewegungsdaten des Segwayroboters trainiert und evaluiert. Das beste erlernte Modell wurde validiert, indem ein analytisch aufgestelltes Bewegungsmodell für einen klassischen Roboter mit Differentialantrieb als Referenz eingesetzt wurde. Zur Validierung des erlernten Modells wurde die Genauigkeit von beiden Bewegungsmodellen an den aufgenommenen Bewegungsdaten untersucht. Das erlernte Modell ist im Durchschnitt bei der Bestimmung der Position des Roboters im nächsten Zeitschritt um 59,48 % genauer und in der Bestimmung der neuen Orientierung des Roboters um 24,61% genauer als das analytisch aufgestellte Bewegungsmodell.

Abstract

The *Segway Loomo* is a self-balancing segway robot, which is constantly balanced by an internal control system. A local path planning strategy was developed in advance for this robot. For local path planning, a motion model of the robot is needed to determine the effect of velocity commands on the robot's pose. In the implemented local path planner, a simple motion model of the robot is used, which does not model the effect of the segway robot's internal control on its motion. In this work, it was investigated whether a more accurate motion model for the *Segway Loomo* robot can be learned by using artificial neural networks to improve the local path planning for this robot. For this purpose, different architectures of feedforward networks were tested. The neural networks were trained and evaluated using recorded motion data of the segway robot. The best learned model was validated by using a standard differential drive motion model as a reference. For the validation of the learned model, the accuracy of both motion models was examined on the recorded motion data. On average, the learned model is 59.48 % more accurate in determining the position of the robot at the next time step and 24.61 % more accurate in determining the new orientation of the robot than the differential drive motion model.

Inhaltsverzeichnis

1

Einleitung

Der *Segway Loomo* Roboter von *SegwayRobotics* ist ein selbstbalancierender Segwayroboter, der sowohl durch einen Fahrer gesteuert als auch autonom fahren kann. Der Segwayroboter ist mit mehreren Sensoren ausgestattet, um mit Menschen und der Umwelt zu interagieren. Der Roboter verfügt unter anderem über eine Kamera zur Umgebungswahrnehmung, Mikrofone zur Ermöglichung von Spracherkennung und Sensoren für die Bestimmung der Position und Orientierung des Roboters durch Odometrie. Mit dem *Software Development Kit* (SDK) ist es außerdem möglich, eigene Anwendungen für den Roboter zu programmieren, indem zum Beispiel auf die Sensordaten zugegriffen werden und Geschwindigkeitsbefehle an die Motoren des Roboters gesendet werden können.

Der *Segway Loomo* Roboter soll als Serviceroboter in Krankenhäusern eingesetzt werden. Er soll unter anderem als automatisierter Guide für Besucher und Patienten dienen und diese zu gewünschten Zielstationen führen. Hierfür wurden Pfadplanungsstrategien für den Segwayroboter entwickelt, damit dieser durch das Krankenhausgebäude navigieren kann. Während sich der Roboter im Krankenhaus bewegt, müssen Kollisionen mit beweglichen Hindernissen wie Menschen vermieden werden. Dafür wird ein lokaler Pfadplaner eingesetzt, der die optimalen Befehlsgeschwindigkeiten bestimmt, die an die Motoren des Roboters gesendet werden, sodass sich der Roboter möglichst nah an einen Zielpunkt heranbewegt und dabei Kollisionen vermeidet. Für die Berechnung der optimalen Befehlsgeschwindigkeiten wird ein Bewegungsmodell des Roboters benötigt, das die Stellung des Roboters angibt, nachdem sich dieser von einer Ausgangsstellung aus mit einer bestimmten Geschwindigkeit über einen gewissen Zeitraum bewegt hat. Bei der bisherigen Implementierung der lokalen Pfadplanung wurde angenommen, dass es sich um einen klassischen Roboter mit Differentialantrieb handelt.

Wie bereits erwähnt, basiert der *Segway Loomo* Roboter auf einer selbstbalancierenden Plattform. Durch eine interne Regelung wird sichergestellt, dass der Segwayroboter stets ausbalanciert ist und nicht nach vorne oder hinten umkippt. Die Auswirkung dieser Regelung auf die Bewegung des Roboters ist schwer zu modellieren und wird nicht von einem Differentialantriebsmodell berücksichtigt. Für die Anwendung des *Segway Loomo* Roboters in einem Krankenhaus ist es besonders wichtig, dass der Roboter beim Navigieren durch das Gebäude keine Kollisionen verursacht. Hierfür wird eine lokale Pfadplanung benötigt, die ein Bewegungsmodell nutzt, das die eigentliche Bewegung des Segwayroboters möglichst genau modelliert. Die lokale Pfadplanung für den *Segway Loomo* Roboter kann also verbessert

werden, indem ein Bewegungsmodell eingesetzt wird, das den Einfluss des ständigen Ausbalancierens des Segwayroboters auf die Bewegung des Roboters berücksichtigt. Aufgrund der Schwierigkeit, ein solches Bewegungsmodell für den *Segway Loomo* Roboter analytisch zu erstellen, bietet es sich an, stattdessen ein Bewegungsmodell mittels maschinellen Lernens zu erlernen.

1.1 Beiträge dieser Arbeit

In dieser Arbeit wurden verschiedene künstliche neuronale Netze eingesetzt, um ein genaues Bewegungsmodell für den *Segway Loomo* Roboter zu erlernen. Hierfür wurden Feedforward-Netze mit einer unterschiedlichen Anzahl an Neuronen in der versteckten Schicht getestet. Außerdem wurden die *tanh* und *Rectifier* Funktionen als Aktivierungsfunktionen für die Neuronen in der versteckten Schicht getestet. Es wurde in dieser Arbeit dargestellt, welche der getesteten Netzwerkarchitekturen für diese Anwendung am besten geeignet ist. Im Durchschnitt haben neuronale Netzwerke mit 40 Neuronen in der versteckten Schicht, die *tanh* als Aktivierungsfunktion verwenden, am besten abgeschnitten.

In dieser Arbeit wurde außerdem das beste erlernte Modell validiert, indem es mit einem analytisch aufgestellten Bewegungsmodell für einen Roboter mit Differentialantrieb verglichen wurde. Das beste in dieser Arbeit erlernte Modell ist im Durchschnitt in der Positionsbestimmung des Roboters um 59,48 % genauer und in der Bestimmung der Orientierung um 24,61 % genauer als das analytisch aufgestellte Modell.

1.2 Verwandte Arbeiten

Es gibt viele Arbeiten, in denen maschinelles Lernen für die lokale Pfadplanung eines mobilen Roboters eingesetzt wird. In (Garrote u. a., 2020) wird beispielsweise ein *Reinforcement Learning*-Ansatz für die lokale Pfadplanung vorgeschlagen. Mittels *Q-Learning* wird die optimale Aktion eines mobilen Roboters bestimmt, welche den maximalen Nutzen erbringt. Der Nutzen wird durch die Nähe zu einer Zieltrajektorie und dem Abstand zu Hindernissen definiert.

Vergleichsweise weniger Arbeiten fokussieren sich darauf, Bewegungsmodelle für mobile Roboter zu erlernen. In (Eliazar und Parr, 2004) wurde eine Methode vorgeschlagen, um die Parameter eines probabilistischen Bewegungsmodells für einen mobilen Roboter zu erlernen. Die Parameter des Bewegungsmodells werden anhand von Sensordaten mit einem EM-Algorithmus (expectation maximization) erlernt. In (Gu und Hu, 2002) wurde ein neuronales Netzwerk eingesetzt, um für eine modellprädiktive Regelung das nichtlineare Bewegungsmodell eines fahrzeugähnlichen mobilen Roboters mit vier Rädern zu erlernen. Das Erlernen von Bewegungsmodellen für mobile Roboter beschränkt sich in der Literatur nicht nur auf autonome Landfahrzeuge. In (Wehbe, Hildebrandt und Kirchner, 2017) wurden mehrere Verfahren des maschinellen Lernens untersucht, um ein Bewegungsmodell für ein autonomes Unterwasserfahrzeug zu erlernen. Es wurden unter anderem neuronale Netze, Support Vector Machines und Gaußprozesse eingesetzt.

1.3 Aufbau dieser Arbeit

Diese Arbeit besteht aus drei Hauptkapiteln. In Kapitel 2 werden die Grundlagen für diese Arbeit erläutert. Zunächst wird gezeigt, wie ein Bewegungsmodell für den *Segway Loomo* Roboter analytisch aufgestellt werden kann. Dieses analytisch aufgestellte Modell wird als Referenz eingesetzt, um das erlernte Bewegungsmodell zu validieren. Dann werden die grundlegenden Aspekte von künstlichen neuronalen Netzen vorgestellt.

Kapitel 3 beschreibt die in dieser Arbeit angewandten Methoden. Dieses Kapitel ist in drei Abschnitte unterteilt, in denen die Arbeitschritte beim Einsatz von neuronalen Netzen für das Erlernen eines Bewegungsmodells dargestellt werden. Zuerst wird beschrieben, wie die für das Erlernen von einem Bewegungsmodell des Segwayroboters notwendigen Daten erfasst wurden. Danach wird darauf eingegangen, wie die erfassten Daten bearbeitet werden müssen, bevor sie effektiv für das Trainieren der neuronalen Netze benutzt werden können. Im letzten Abschnitt dieses Kapitels werden die in dieser Arbeit eingesetzten neuronalen Netze vorgestellt. Es wird darauf eingegangen, wie die eingesetzten neuronalen Netze aufgebaut sind und welche Lernstrategie für das Trainieren dieser Netze verwendet wurde.

Abschließend werden in Kapitel 4 die erlernten Modelle evaluiert. Es wird in diesem Kapitel dargestellt, welche neuronalen Netze die besten Bewegungsmodelle erlernen. Außerdem wird in diesem Kapitel die Genauigkeit des besten erlernten Modells mit der des analytisch aufgestellten Bewegungsmodells aus Kapitel 2 verglichen.

2

Grundlagen

In diesem Kapitel werden zwei grundlegende Konzepte dieser Arbeit erläutert. Zuerst wird beschrieben, wie ein Bewegungsmodell für den Segwayroboter analytisch aufgestellt werden kann. Dieses Bewegungsmodell wird als Maßstab dienen, an dem das erlernte Bewegungsmodell gemessen wird. Danach werden die Grundlagen von künstlichen neuronalen Netzen erläutert, welche in dieser Arbeit eingesetzt werden, um ein Bewegungsmodell für den Segwayroboter zu erlernen.

2.1 Differentialantriebsmodell

Das Ziel dieser Arbeit ist es, ein Bewegungsmodell des *Loomo Segway* Roboters zu erlernen, das dafür eingesetzt werden kann, die für diesen Roboter entwickelte lokale Pfadplanung zu verbessern. Der lokale Pfadplaner geht von einem einfachen Roboter mit Differentialantrieb aus. Damit die Pfadplanung verbessert werden kann, muss das erlernte Modell also die Bewegung des Segwayroboters besser modellieren als ein Bewegungsmodell für einen Roboter mit Differentialantrieb. Aus diesem Grund wird in diesem Abschnitt ein Differentialantriebsmodell hergeleitet, das als Referenz genutzt wird, um die Genauigkeit des erlernten Models zu überprüfen. Das in diesem Abschnitt hergeleitete Differentialantriebsmodell basiert auf dem in (Thrun, Burgard und Fox, 2005) beschriebenen probabilistischem Bewegungsmodell.

Dieses Bewegungsmodell gibt die Stellung eines Roboters mit Differentialantrieb an, wenn dieser von einer Ausgangsstellung aus bestimmte Steuergeschwindigkeiten über einen gewissen Zeitraum ausführt. Die Stellung des Roboters wird, wie in Abbildung 2.1 dargestellt ist, durch dessen Position und Orientierung im zweidimensionalen Raum in Bezug auf ein Referenzkoordinatensystem angegeben. Mit dem Roboter ist ein lokales Koordinatensystem verbunden, wobei sich dessen Ursprung im Mittelpunkt des Roboters befindet und die x-Achse des Koordinatensystems in die Richtung der Roboterfront zeigt. Die Position des Roboters wird durch die Koordinaten x und y des Ursprungs des lokalen Koordinatensystems in dem globalen Referenzystem beschrieben. Die Orientierung des Roboters wird durch den Winkel θ zwischen den x-Achsen des lokalen und globalen Koordinatensystem definiert. Die Stellung des Roboters $\mathbf{x_t}$ zu einem Zeitpunkt t wird dann durch den folgenden

Vektor angegeben:

$$\mathbf{x}_t = \begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} \tag{2.1}$$

Der *Loomo Segway* Roboter wird gesteuert, indem eine Translationsgeschwindigkeit v und

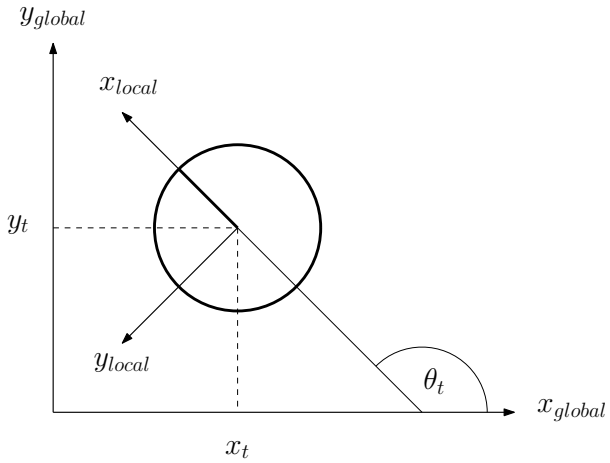


Abbildung 2.1: Diese Abbildung zeigt, wie die Stellung des Roboters definiert ist.

eine Rotationsgeschwindigkeit ω vorgegeben werden. Der Steuerbefehl $\mathbf{u}_t$ zum Zeitpunk t wird daher durch den folgenden Vektor angegeben:

$$\mathbf{u}_t = \begin{bmatrix} v_t \\ \omega_t \end{bmatrix} \tag{2.2}$$

Das Bewegungsmodell bestimmt die Stellung $\mathbf{x}_{t+1}$ des Roboters zum Zeitpunk $t{+}1$ am Ende einer Bewegung, bei der von einer Anfangsstellung $\mathbf{x}_t$ aus die Geschwindigkeiten $\mathbf{u}_t$ für den Zeitraum $\Delta t = (t,\, t{+}1]$ konstant gehalten werden:

$$\mathbf{x}_{t+1} = f\left(\mathbf{x}_t, \mathbf{u}_t\right) \tag{2.3}$$

Die Art von Bewegung, die der Roboter in diesem Zeitraum vollführt, ist von der Rotationsgeschwindigkeit ω_t abhängig. Bei $\omega_t = 0$ führt der Roboter eine geradlinige Bewegung ohne Richtungsänderung aus und bei $\omega_t \neq 0$ vollführt der Roboter eine Kreisbewegung.

Es wird zuerst die lineare Bewegung des Roboters betrachtet. Wenn der Roboter für den Zeitraum Δt mit der konstanten Geschwindigkeit v_t fährt, legt er eine Strecke von $v_t\, \Delta t$ zurück. Die Koordinaten des Roboters am Ende dieser Bewegung können per Trigonometrie bestimmt werden. Dabei wird die zurückgelegte Strecke auf die Achsen des Weltkoordinatensystems projeziert und dann mit den entsprechenden Koordinaten der Ausgangsposition des Roboters aufaddiert. Da es sich um eine lineare Bewegung handelt, verändert sich die Orientierung des Roboters nicht. Die Stellung des Roboters am Ende der linearen Bewegung ist demnach:

$$\mathbf{x}_{t+1} = \begin{bmatrix} x_t + \cos(\theta_t)\, v_t\, \Delta t \\ y_t + \sin(\theta_t)\, v_t\, \Delta t \\ \theta_t \end{bmatrix} \tag{2.4}$$

Wenn der Roboter mit einer konstanten Translationsgeschwindigkeit v_t und einer konstanten Rotationsgeschwindigkeit $\omega_t \neq 0$ angetrieben wird, bewegt sich dieser, wie in Abbildung 2.2 zu sehen ist, entlang eines Kreises mit folgendem Radius r:

$$r = \left| \frac{v_t}{\omega_t} \right| \tag{2.5}$$

Die Koordinaten $\begin{bmatrix} x_c & y_c \end{bmatrix}^T$ des Mittelpunktes dieses Kreises können bestimmt werden,

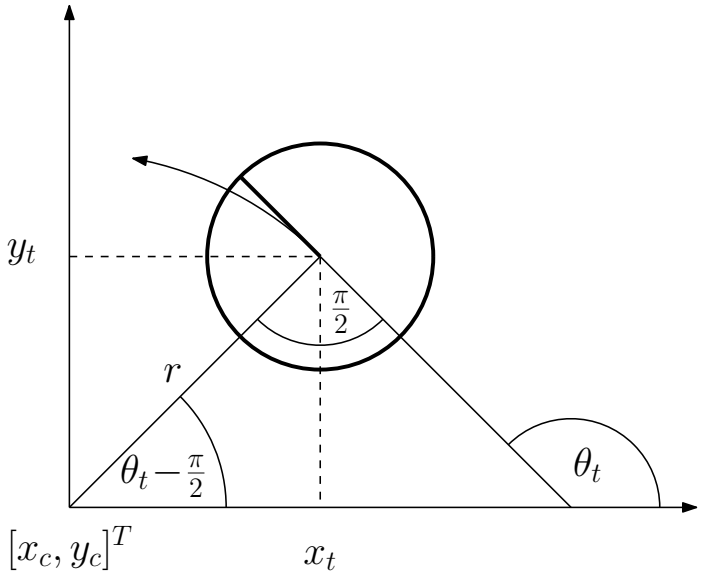


Abbildung 2.2: Diese Abbildung zeigt die Stellung des Roboters in Relation zu dem Mittelpunkt $\begin{bmatrix} x_c & y_c \end{bmatrix}^T$ eines Kreises, wenn der Roboter um diesen eine Kreisbewegung durchführt.

indem die Projektionen des Radius auf die Achsen des Weltkoordinatensystems von den aktuellen Koordinaten des Roboters subtrahiert werden. Hierbei ist zu beachten, dass der Winkel zwischen der x-Achse des Weltkoordinatensystems und der Verbindungslinie zwischen Kreismittelpunkt und der Position des Roboters auf der Kreisbahn $\theta_t - \frac{\pi}{2}$ beträgt. Mit den trigonometrischen Identitäten $\cos\left(\theta_t - \frac{\pi}{2}\right) = \sin\left(\theta_t\right)$ und $\sin\left(\theta_t - \frac{\pi}{2}\right) = -\cos\left(\theta_t\right)$ folgt dann:

$$\begin{bmatrix} x_c \\ y_c \end{bmatrix} = \begin{bmatrix} x_t - \frac{v_t}{\omega_t} \sin(\theta_t) \\ y_t + \frac{v_t}{\omega_t} \cos(\theta_t) \end{bmatrix} \tag{2.6}$$

Nach einer Bewegung von der Dauer Δt hat der Roboter eine Strecke von $v\,\Delta t$ entlang der Kreisbahn zurückgelegt und seine Ausrichtung um $\omega\,\Delta t$ verändert. Die neuen Koordinaten des Roboters $\begin{bmatrix} x_{t+1} & y_{t+1} \end{bmatrix}^T$ auf der Kreisbahn am Ende dieser Bewegung können von dem Mittelpunkt des Kreises ausgehend per Trigonometrie bestimmt werden:

$$\begin{bmatrix} x_{t+1} \\ y_{t+1} \end{bmatrix} = \begin{bmatrix} x_c + \frac{v_t}{\omega_t} \sin(\theta_t + \omega_t\,\Delta t) \\ y_c - \frac{v_t}{\omega_t} \cos(\theta_t + \omega_t\,\Delta t) \end{bmatrix} \tag{2.7}$$

Die Stellung des Roboters $\mathbf{x_{t+1}}$ zum Zeitpunkt $t+1$ am Ende einer Kreisbewegung für die Dauer Δt ausgehend von einer Startstellung $\mathbf{x}_t$ zum Zeitpunkt t wird demnach wie folgt

berechnet:

$$\mathbf{x}_{t+1} = \begin{bmatrix} x_t - \frac{v_t}{\omega_t}\sin(\theta_t) + \frac{v_t}{\omega_t}\sin(\theta_t + \omega_t \Delta t) \\ y_t + \frac{v_t}{\omega_t}\cos(\theta_t) - \frac{v_t}{\omega_t}\cos(\theta_t + \omega_t \Delta t) \\ \theta_t + \omega_t \Delta t \end{bmatrix} \tag{2.8}$$

2.2 Künstliche neuronale Netze

Künstliche neuronale Netze sind ein Teil des maschinellen Lernens. Das maschinelle Lernen umfasst Algorithmen, die anhand von Beispieldaten Wissen generieren und basierend auf diesem Wissen Vorhersagungen oder Entscheidungen für neue Daten treffen. Hierfür erkennen die Algorithmen Muster und Zusammenhänge in den sogenannten Trainingsdaten und erstellen auf der Grundlage von diesen Erkenntnissen ein Modell auf, das dann genutzt werden kann, um auch zuvor ungesehene Daten zu bewerten. Neuronale Netze werden meistens im Zusammenhang mit dem überwachten Lernen eingesetzt. Beim überwachten Lernen bestehen die Daten aus Paaren von Eingabewerten und vorgegebenen Ausgabewerten. Das Ziel ist es, eine Funktion zu erlernen, welche die Eingabewerte möglichst genau auf die gewünschten Ausgabewerte abbildet.

Neuronale Netze bestehen aus mehreren künstlichen Neuronen. Ein künstliches Neuron erhält ein oder mehrere Eingabewerte, verarbeitet diese und produziert ein entsprechendes Ausgabesignal. Dabei wird jeder Eingabewert mit einem Gewichtungsfaktor versehen und diese gewichteten Eingaben werden dann aufsummiert. Anschließend wird dieser Summe ein Bias-Wert hinzugefügt. Die gewichtete Summe der Eingabewerte addiert mit dem Bias-Wert wird dann einer Aktivierungsfunktion übergeben. Es können verschiedene Funktionstypen als Aktivierungsfunktion für ein künstliches Neuron eingesetzt werden. Die Ausgabe der Aktivierungsfunktion ist schließlich auch das Ausgabesignal a des künstlichen Neurons:

$$a = \varphi\left(\sum_{j=0}^{n} w_j x_j\right) \tag{2.9}$$

wobei φ die Aktivierungsfunktion, x_j die n Eingabewerte und w_j die entsprechenden Gewichtungsfaktoren sind. Der Bias-Wert w_0 wird als Gewichtung eines zusätzlichen konstanten Eingabewerts $x_0 = 1$ definiert.

Ein neuronales Netz ist aus aufeinanderfolgenden Schichten von künstlichen Neuronen aufgebaut, welche miteinander verbunden sind. Bei einem Feedforward-Netz, der einfachsten Form eines neuronalen Netzes, sind diese Schichten so miteinander verbunden, dass jedes Neuron aus einer Schicht die Ausgabesignale von allen Neuronen der vorangegangenen Schicht als Eingabewerte erhält und sein Ausgabesignal an alle Neuronen der darauffolgenden Schicht als Eingabewert weitergibt. Die Eingabewerte für das neuronale Netz an sich werden in der sogenannten Eingabeschicht an die erste Schicht von Neuronen übergeben. Auf die Eingabeschicht folgen eine beliebige Anzahl an Schichten aus Neuronen, welche versteckte Schichten gennant werden. Die letzte Schicht aus künstlichen Neuronen des neuronalen Netzwerks wird Ausgabeschicht genannt. Die Ausgaben der Neuronen dieser Schicht

sind auch die Ausgabewerte des gesamten neuronalen Netzes. Die Anzahl der Neuronen der Ausgabeschicht entspricht deshalb der Anzahl an Ausgabewerten des neuronalen Netzes. Die beschriebene Topologie eines Feedforward-Netzes wird in Abbildung 2.3 veranschaulicht.

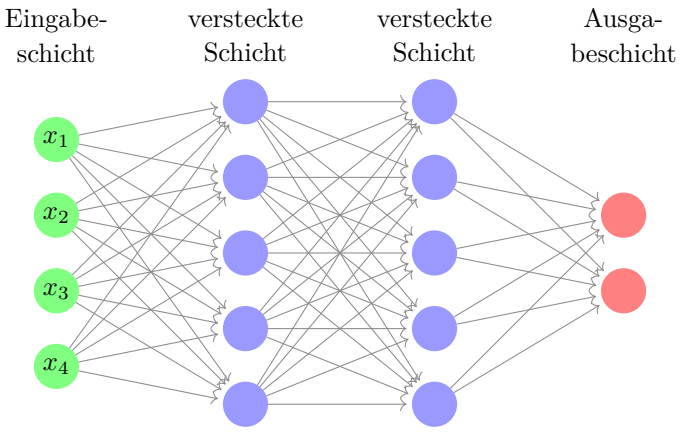


Abbildung 2.3: Diese Abbildung stellt die Topologie eines Feedforward-Netzes mit vier Eingabewerten, zwei versteckten Schichten aus jeweils fünf künstlichen Neuronen und einer Ausgabeschicht aus zwei Neuronen dar.

Im Zusammenhang des überwachten Lernens wird ein künstliches neuronales Netz eingesetzt, um anhand von Daten bestehend aus Paaren von Eingabewerten und Zielwerten ein Modell zu erlernen, welches die Eingabewerte möglichst genau auf die gewünschten Zielwerte abbildet. Hierfür wird ein Teil der Daten als Trainingsdaten verwendet, an denen das neuronale Netz das Modell erlernt. Der Rest der Daten wird verwendet, um das erlernte Modell an neuen, nicht im Training verwendeten Daten zu validieren.

Ein neuronales Netz wird iterativ an den Trainingsdaten angelernt. Zunächst werden die Gewichte und Bias-Werte des neuronalen Netzes zufällig initialisiert. Dann wird für jedes Paar aus Eingabewerten und dazugehörigen Zielwerten der Testdaten die Ausgaben des neuronalen Netzes anhand der Eingabewerte berechnet und anschließend mit den entsprechenden Zielwerten verglichen. Die Ausgabewerte eines neuronalen Netzes werden bestimmt, indem zuerst die Ausgabesignale der Neuronen aus der ersten versteckten Schicht anhand der Eingabewerte des neuronalen Netzes berechnet werden. Basierend auf diesen Ausgaben können die Ausgabesignale der Neuronen aus der nächsten Schicht bestimmt werden. Die Ausgabewerte der Neuronen einer Schicht werden dann iterativ anhand der Ausgaben der Neuronen aus der jeweiligen vorherigen Schicht bestimmt, bis schließlich die Ausgaben der Neuronen der Ausgabeschicht berechnet werden, welche die Ausgabewerte des neuronalen Netzes sind.

Das Ausgabesignal $a_i^{(k)}$ von dem i-ten Neuron der Schicht k kann wie folgt anhand der Ausgaben der n Neuronen aus der vorherigen Schicht $k-1$ bestimmt werden:

$$a_i^{(k)} = \varphi^{(k)}\left(net_i^{(k)}\right) = \left(\sum_{j=0}^{n} w_{ij}^{(k)} a_j^{(k-1)}\right) \tag{2.10}$$

wobei $w_{ij}^{(k)}$ der Gewichtungsfaktor des i-ten Neurons aus der k-ten Schicht ist, welcher

das Ausgabesignal $a_j^{(k-1)}$ des j-ten Neurons aus der Schicht $k-1$ gewichtet und $\varphi^{(k)}$ die Aktivierungsfunktion darstellt, welche bei den Neuronen der k-ten Schicht eingesetzt wird. Der Bias-Wert $w_{i0}^{(k)}$ des i-ten Neurons aus der Schicht k wird als Gewichtungsfaktor der konstanten Ausgabe $a_0^{(k-1)} = 1$ eines zusätzlichen Neurons definiert.

Um den Fehler des neuronalen Netzes bei der Bestimmung der Zielwerte zu ermitteln, wird eine Kostenfunktion $e\left(f\left(\mathbf{w}, \mathbf{x}\right), \mathbf{y}\right)$ eingesetzt, wobei $\mathbf{y}$ die Zielwerte und $f\left(\mathbf{w}, \mathbf{x}\right)$ die Ausgaben des neuronalen Netzes in Abhängigkeit von den Gewichten und Bias-Werten $\mathbf{w}$ und den Eingabewerten $\mathbf{x}$ sind. Der Fehler des neuronalen Netzes kann auch über mehrere Paare von Eingabe- und Zielwerten bestimmt werden.

Nachdem der Fehler des neuronalen Netzes errechnet wurde, werden die Gewichte und Bias-Werte des Netzes so angepasst, dass der Fehler verringert wird. Dabei wird der Fehler des neuronalen Netzes von der Ausgabeschicht ausgehend bis zur ersten versteckten Schicht auf die einzelnen Neuronen des Netzes und deren Gewichtungsfaktoren zurückgeführt. Aus diesem Grund wird das Lernverfahren eines neuronalen Netzes als Fehlerrückführung bezeichnet. Das Bestimmen der Netzwerkausgaben, die Berechnung des Fehlers anhand der Kostenfunktion und das anschließende Anpassen der Gewichte und Bias-Werte wird für mehrere Iterationen durchgeführt.

Eine Variante der Fehlerrückführung ist das Gradientenverfahren. Bei diesem Verfahren wird für die Anpassung der Gewichte der Gradient der Fehlerfunktion $\nabla e\left(\mathbf{w}\right)$ in Abhängigkeit von den aktuellen Gewichten und Bias-Werten des Netzwerkes bestimmt. Dieser Gradient ist ein Vektor, der die partiellen Ableitungen der Fehlerfunktion nach allen Gewichten und Bias-Werten des neuronalen Netzes $\frac{\partial e}{\partial w_{ij}^{(k)}}$ beinhaltet. Der Gradient kann als ein Vektor aufgefasst werden, welcher ausgehend von den aktuellen Gewichten und Bias-Werten des Netzes in die Richtung des steilsten Anstiegs der Fehlerfunktion zeigt. Die Idee des Gradientenverfahrens ist es, die Kostenfunktion zu verringern, indem in jeder Iteration die Gewichte und Bias-Werte des neuronalen Netzes so aktualisiert werden, dass ein Schritt in Richtung des negativen Gradienten getan wird. Die Gewichte und Bias-Werte des Netzwerks werden bei diesem Verfahren bei jeder Iteration nach folgender Aktualisierungsregel angepasst:

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \mu \nabla e\left(\mathbf{w}_n\right) \tag{2.11}$$

wobei μ eine Schrittweite ist und $\mathbf{w}_n$ die Gewichte und Bias-Werte des neuronalen Netzes zur Iteration n bezeichnen.

3

Methoden

In diesem Kapitel wird die Vorgehensweise zum Erlernen eines Bewegungsmodells für den *Loomo Segway* Roboter beschrieben. Hierfür wurde in drei Arbeitsschritten vorgegangen: Zuerst wurden die für das Erlernen des Bewegungsmodells notwendigen Daten aufgenommen. Dann wurden die aufgenommenen Daten aufbereitet, sodass sie effektiv für das Lernen eingesetzt werden können. Zuletzt wurden die künstlichen neuronalen Netze erstellt, welche für diese Aufgabe eingesetzt wurden.

3.1 Aufnahme von Bewegungsdaten

Das Ziel dieser Arbeit ist es, ein Bewegungsmodell für den Loomo Segway Roboter zu erlernen. Um ein Bewegungsmodell zu erlernen, werden Daten benötigt, welche die Kinematik des Roboters beschreiben. Diese Daten werden auch als Referenz benötigt, um die Genauigkeit des erlernten Modells und des analytisch aufgestellten Bewegungsmodells anhand der aufgenommenen Bewegungsdaten zu ermitteln. Aus diesem Grund wurde im Zuge dieser Arbeit eine Datenaufnahme durchgeführt, bei welcher der Segwayroboter auf einer freien Fläche bewegt wurde und währenddessen verschiedene Daten über diese Bewegung erfasst wurden.

Um die Position und Ausrichtung des Roboters zu erfassen, wurde zum einen das Trackingsystem von *OptiTrack* verwendet. Dieses Trackingsystem kann die Position von einem mit retroreflektierenden Sensoren markierten Objekt bei guten Aufnahmebedingungen mit einer Genauigkeit im Submillimeterbereich verfolgen (NaturalPoint Inc., 2021). Um die Bewegung des Roboters mit dem Trackingsystem zu verfolgen, wurden vier Kameras des Trackingsystems an den Ecken der quadratischen Fläche, auf welcher der Roboter bewegt wurde, aufgestellt. So decken die Kameras den gesamten Bewegungsbereich des Roboters aus unterschiedlichen Winkeln ab. Es wurden dann vier Sensoren an dem Kopf des Segwayroboters befestigt, sodass die Kameras die Sensoren gut erfassen können. Dies ist ist in Abbildung 3.1 visualisiert. Den Sensoren wird ein lokales Koordinatensystem mit Ursprung in der geometrischen Mitte der Sensoren zugewiesen. Das Trackingsystem nimmt die Position und Ausrichtung dieses Koordinatensystems in Bezug auf ein globales Koordinatensystem im dreidimensionalen Raum auf. Die Orientierung wird als Quaternion angegeben.

Das Trackingsystem ist mit ROS kompatibel und die erfassten Stellungsdaten können in Echtzeit als ROS-Nachrichten gesendet werden.

Abbildung 3.1: Dieses Bild zeigt den Segwayroboter mit am Roboterkopf befestigten Sensoren und zwei Kameras des Trackingsystems im Hintergrund

Während der Erfassung der Stellung des Segwayroboters durch das Trackingsystem wurde der Roboter mit einer App gesteuert. Mithilfe dieser App können die Translations- und Rotationsgeschwindigkeiten des Roboters vorgegeben werden, indem diese als Befehlssignale an den Roboter gesendet werden. Die gesendeten Geschwindigkeitssignale können ebenfalls als ROS-Nachrichten gesendet und aufgenommen werden.

Mit Sensoren an den Radmotoren des Segwayroboters kann die Position und Ausrichtung des Roboters auch per Odometrie in Echtzeit bestimmt werden. Die aktuelle Position und Ausrichtung im dreidimensionalen Raum wird in Bezug auf eine Ausgangsstellung des Roboters angegeben. Die Ausrichtung des Roboters wird hier wie bei dem Trackingsystem als Quaternion angegeben. Die Stellungsdaten der Odometrie können von der API des Roboters auch als ROS-Nachrichten in Echtzeit gesendet werden.

Insgesamt wurde die Bewegung des Segwayroboters für vier Stunden aufgezeichnet. Während der Bewegung des Roboters wurden die durch das Trackingsystem und der Odometrie des Roboters erfassten Daten sowie die Signale der App, mit welcher der Roboter gesteuert wurde, als ROS-Nachrichten aufgenommen. Das Trackingsystem gibt die Stellung des Roboters mit einer Frequenz von 75 Hz an, die Odometrie des Roboters mit 15 Hz und die App sendet Steuersignale mit einer Frequenz von 12,5 Hz. Es wurden im Ganzen 1 075 513 Nachrichten des Trackingsystems, 215 115 Nachrichten der Odometrie und 172 061 Nachrichten der App aufgenommen. Während der Aufnahme wurden verschiedene Bewegungsabläufe wie starkes Abbremsen, enge und weitläufige Kurven sowie Rückwärts-

fahren ausgeführt, damit die aufgenommenen Daten repräsentativ für verschiedenste Bewegungsabläufe sind. So können die aufgenommenen Daten als gute Basis dienen, um ein Bewegungsmodell zu erlernen.

3.2 Aufbereitung der Bewegungsdaten

Wie im vorigen Abschnitt beschrieben, wurden verschiedene Daten über die Bewegung des Roboters aufgenommen. Anhand dieser Daten soll ein Bewegungsmodell des Segwayroboters erlernt werden und anschließend die Genauigkeit des erlernten Modells und die des aufgestellten Differentialantriebsmodells ermittelt werden. Um für diese Anwendungen effektiv eingesetzt werden zu können, müssen die aufgenommenen Daten jedoch vorher aufbereitet werden.

Bei der Datenaufnahme wurden Daten aus verschiedenen Quellen aufgenommen: dem Trackingsystem, der Roboter API, welche die Stellungsdaten der Odometrie wiedergibt, und der App, die Steuerbefehle an den Roboter sendet. Diese drei Quellen senden Daten mit unterschiedlichen Frequenzen. Die aufgenommenen Daten geben dadurch Informationen über die Bewegung des Roboters zu unterschiedlichen Zeitpunkten. Aus diesem Grund wurden die Daten durch lineare Interpolation an einheitlichen Zeitpunkten, gegeben durch die mit der niedrigsten Frequenz gesendeten Daten, ausgewertet. Diese lineare Interpolation der Daten wird in Abbildung 3.2 beispielhaft anhand der aufgenommenen x-Positionswerte des Trackingsystems dargestellt. Die Daten geben so zu insgesamt 172 061 Zeitpunkten jeweils eine Angabe zur gemessenen Stellung des Roboters durch das Trackingsystem, der per Odometrie ermittelten Stellung des Roboters und den Steuergeschwindigkeiten.

Das Trackingsystem und die Odometrie geben die Stellung des Roboters in Bezug auf unterschiedlich definierte Koordinatensysteme an. Damit die Stellungsdaten aus beiden Quellen für das Erlernen eines Bewegungsmodells genutzt werden können, müssen sie vorher so transformiert werden, dass sie die Stellung des Roboters in Bezug auf dasselbe Koordinatensystem angeben. Wie in Abbildung 3.3 zu sehen ist, wird bei der Odometrie für die Bestimmung der Stellung des Roboters ein Koordinatensystem benutzt, das auf dieselbe Weise definiert ist wie das lokale Koordinatensystem des Roboters aus dem in Abschnitt 2.1 aufgestellten Differentialantriebsmodell. Der einzige Unterschied ist, dass bei der Odometrie die Stellung des Roboters im dreidimensionalen Raum angegeben wird. Aus diesem Grund wird eine Transformation bestimmt, welche die Stellungsdaten des Trackingsystems in das von der Odometrie und dem Differentialantriebsmodell benutzte Koordinatensystem transformiert. Um diese Transformation zu bestimmen, werden die Rotationsmatrix $\mathbf{R}$ und der Translationsvektor $\mathbf{t}$ berechnet, welche die Positionsdaten des Trackingsystems auf die Positionsdaten der Odometrie abbilden:

$$\mathbf{o}_i = \mathbf{R}\,\mathbf{t}_i + \mathbf{t} + \mathbf{e} \tag{3.1}$$

wobei $\{\mathbf{t}_i\}, i = 1, 2, ...n$ die Menge der durch das Trackingsystem aufgenommenen Positionsdaten zu jedem Zeitpunkt i und $\{\mathbf{o}_i\}$ die Menge der entsprechenden durch die Odometrie bestimmten Positionen zu denselben Zeitpunkten ist. Der Vektor $\mathbf{e}$ stellt Fehlerwerte dar,

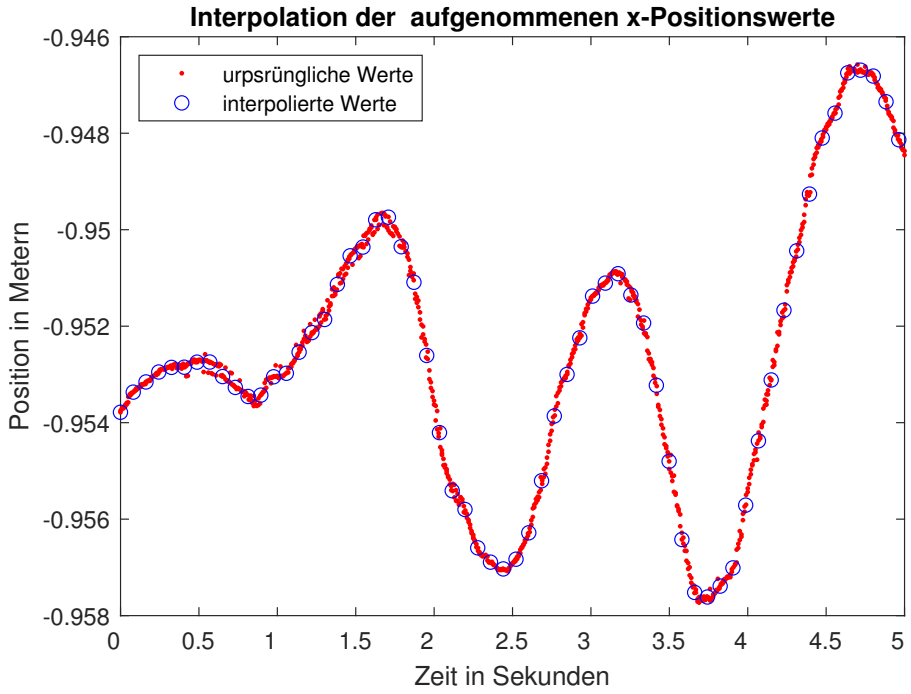


Abbildung 3.2: Diese Abbildung zeigt die durch das Trackingsystem aufgenommenen x-Koordinaten des Roboters in rot und die zu den einheitlichen Zeitpunkten interpolierten Werte in blau.

die aufgrund der unterschiedlichen Genauigkeit der Messungen der Position des Roboters auftreten.

Die Positionsbestimmung durch die Odometrie wird aufgrund sich aufaddierender Fehler über eine längere Strecke immer ungenauer, wodurch die Positionsdaten der Odometrie immer stärker von den eigentlichen Positionen des Roboters abweichen. Aufgrund dieser immer größer werdenden Abweichungen in den Positionsdaten der Odometrie kann keine einzelne Transformation bestimmt werden, welche die gesamten Positionsdaten des Trackingsystems genau nach den Positionsdaten der Odometrie ausrichtet. Daher wird nicht eine Transformation bestimmt, welche auf die gesamten Stellungsdaten des Trackingsystems angewendet wird, sondern es wird iterativ für jede Stellung eine Transformation bestimmt. Hierfür wird für jede Stellung, die zum Zeitpunkt j vom Trackingsystem aufgenommen wurde, eine Transformation bestimmt, welche die drei Positionsdaten $\mathbf{t}_{j-1}$, $\mathbf{t}_j$ und $\mathbf{t}_{j+1}$ des Trackingsystems auf die entsprechenden Positionsdaten $\mathbf{o}_{j-1}$, $\mathbf{o}_j$ und $\mathbf{o}_{j+1}$ der Odometrie abbildet. Die Transformation wird also anhand von drei Paaren von Positionsdaten berechnet. Dies ist die Mindestanzahl an Paaren von Positionen, um eine Transformation zu bestimmen, welche die Ausrichtung der Positionsdaten des Trackingsystems zueinander beibehält. Indem für die Bestimmung der Transformation nur drei aufeinanderfolgende Datenpunkte benutzt werden, sind die Abweichungen in den drei Positionsdaten der Odometrie durch sich aufaddierende Fehler gering. Dadurch können die drei Positionsdaten des Trackingsystems durch die Transformation genau auf die entsprechenden Odometriedaten abgebildet werden.

Um diese Transformationen zu bestimmen, wurde das in (Arun, Huang und Blostein, 1987) beschriebene Verfahren benutzt. Mit diesem Verfahren werden die Rotationsmatrix $\mathbf{R}$

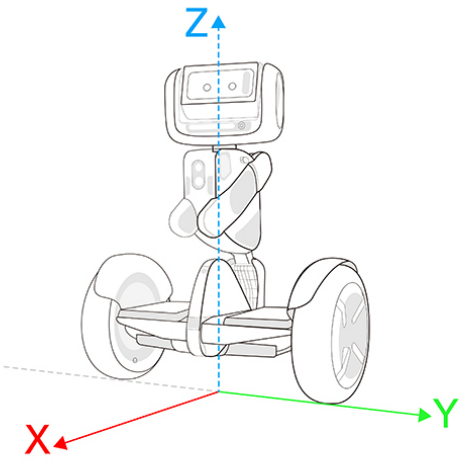


Abbildung 3.3: Diese Abbildung zeigt das Referenzkoordinatensystem des Segwayroboters, welches bei der Stellungsbeschreibung des Roboters durch die Odometrie verwendet wird. (SegwayRobotics Inc., 2019)

und der Translationsvektor $\mathbf{t}$ bestimmt, welche die folgende Fehlergleichung mit der Methode der kleinsten Fehlerquadrate minimieren:

$$\varepsilon^2 = \sum_{i=j-1}^{j+1} \|\mathbf{o}_i - (\mathbf{R}\,\mathbf{t}_i + \mathbf{t})\|^2 \tag{3.2}$$

Mit der auf diese Weise bestimmten Transformation können die vom Trackingsystem aufgenommenen Stellungsdaten des Roboters zu jedem Zeitpunkt j in das Koordinatensystem, welches von der Odometrie und dem Differentialantriebsmodell benutzt wird, transformiert werden. Die Positionsdaten werden wie folgt transformiert:

$$\mathbf{t}'_j = \mathbf{R}\,\mathbf{t}_j + \mathbf{t} \tag{3.3}$$

Die Orientierung des Roboters zu einem Zeitpunkt j wird vom Trackingsystem als Quaternion Q_j angegeben. Um diese Quaternion in das Koordinatensystem der Odometrie zu transformieren, wird die berechnete Rotationsmatrix $\mathbf{R}$ in eine Quaternion Q umgewandelt:

$$Q'_j = Q\,Q_j \tag{3.4}$$

In jeder Iteration werden die transformierten Stellungswerte zu dem Zeitpunkt j einer Menge der Ausgangsstellungen hinzugefügt und die darauffolgenden Stellungswerte zu dem Zeitpunkt $j+1$ in einer Menge der Nachfolgestellungen gespeichert. So wird sicher gestellt, dass jedes Paar von Ausgangsstellungs- und Nachfolgestellungswerten mit derselben Rotationsmatrix und demselben Translationsvektor transformiert wurden.

Auch die durch die Odometrie bestimmten Stellungswerte werden in eine Menge aus Ausgangsstellungen und eine Menge aus Nachfolgestellungen unterteilt. Hierfür werden die

Stellungsdaten zu den Zeitpunkten $i = 1, 2, ..., n{-}1$ in die Menge der Ausgangsstellungen gespeichert und die Stellungsdaten zu den Zeitpunkten $k = 2, 3, ..., n$ werden in die Menge der Nachfolgestellungen hinzugefügt. Auf diese Weise werden die aufgenommenen Stellungsdaten des Roboters für das Erlernen des Bewegungsmodell in Eingabe-und Zielwerte aufgeteilt.

Für das Erlernen des Bewegungsmodells wird die Bewegung des Roboters in der zweidimensionalen Ebene betrachtet. Damit die aufgenommenen Stellungsdaten des Roboters für das Erlernen eines solchen Bewegungsmodells benutzt werden können, müssen sie noch so angepasst werden, dass sie die Stellung des Roboters in der zweidimensionalen Ebene darstellen. Hierfür werden von den aufgenommenen Positionsdaten nur die x und y-Koordinaten benutzt. Die Orientierung des Roboters wird von den aufgenommenen Daten als Quaternion angegeben. Um die Ausrichtung des Roboters im zweidimensionalen Raum anzugeben, wurden die Quaternionen zunächst in Eulerwinkel konvertiert. Der Winkel θ, welcher die Rotation um die zum Boden senkrecht stehende z-Achse beschreibt, wird benutzt, um die Ausrichtung des Roboters anzugeben.

3.3 Erlernen von Bewegungsmodellen

In dieser Arbeit wird anhand von aufgenommenen Bewegungsdaten ein Bewegungsmodell für den *Loomo Segway* Roboter erlernt. Neuronale Netze können komplexe, nichtlineare Zusammenhänge modellieren ohne dabei a priori Wissen über diese Zusammenhänge zu benötigen. Aus diesem Grund werden verschiedene neuronale Netze dafür eingesetzt, ein Bewegungsmodell für den *Segway Loomo* Roboter zu erlernen. Hierfür wird die *Deep Learning Toolbox* von MATLAB verwendet.

Die aufbereiteten Daten aus dem vorherigen Abschnitt beschreiben die durch das Trackingsystem erfassten Stellungen des Roboters $\{x'_i\}$, $\{y'_i\}$, $\{\theta'_i\}$, die durch die Odometrie bestimmten Stellungen $\{x_i\}$, $\{y_i\}$, $\{\theta_i\}$ und die Geschwindigkeitsbefehle $\{v_i\}$, $\{\omega_i\}$ an den Roboter zu den einheitlichen Zeitpunkten $i = 1, 2, ..., n{-}1$. Außerdem werden von den aufbereiteten Daten für jede Stellung des Roboters zu einem Zeitpunkt i die jeweils darauffolgende durch das Trackingsystems beziehungsweise der Odometrie erfassten Stellung zum nächsten Zeitpunkt $k = i{+}1$ angegeben. Diese Nachfolgestellungen werden jeweils durch $\{x'_k\}$, $\{y'_k\}$, $\{\theta'_k\}$ und $\{x_k\}$, $\{y_k\}$, $\{\theta_k\}$, $k = 2, 3, ..., n$ beschrieben.

Das neuronale Netz soll anhand der Stellung des Roboters und den Geschwindigkeitsbefehlen an den Roboter zu einem Zeitpunkt t die Stellung des Roboters zu dem darauf folgendem Zeitpunkt $t{+}1$ bestimmen. Dadurch wird die Bewegung des Roboters über den Zeitraum $\Delta t = (t,\, t{+}1]$ erlernt. Die aufgenommenen Daten werden dafür in Eingabe- und Zielwerte aufgeteilt. Die Paare von Stellungsdaten des Trackingsystems und Geschwindigkeitsbefehlen an den Roboter$\{x'_i\}$, $\{y'_i\}$, $\{\theta'_i\}$, $\{v_i\}$, $\{\omega_i\}$ zu den Zeitpunkten $i = 1, 2, ..., n{-}1$ und die Paare von Stellungsdaten der Odometrie und den jeweiligen Geschwindigkeitsbefehlen $\{x_i\}$, $\{y_i\}$, $\{\theta_i\}$, $\{v_i\}$, $\{\omega_i\}$ fungieren als Eingabewerte für das neuronalen Netz. Die Nachfolgestellungen des Trackingsystems $\{x'_k\}$, $\{y'_k\}$, $\{\theta'_k\}$ beziehungsweise die Nachfolgestellungen der Odometrie $\{x_k\}$, $\{y_k\}$, $\{\theta_k\}$ zu den Zeitpunkten $k = 2, 3, ..., n$ sind dann die jeweiligen Zielwerte, die das neuronale Netz bestimmen soll.

Durch diese Wahl von Eingabe- und Zielwerten erhält das neuronale Netz wie gewünscht die Position und Orientierung des Roboters und die Geschwindigkeitsbefehle an den Roboter zu einem Zeitpunkt t als Eingaben und soll anhand dieser Werte die Position und Orientierung des Roboters zu dem darauffolgenden Zeitpunkt $t+1$ bestimmen. Dies wird in Tabelle 3.4 veranschaulicht.

Tabelle 3.4: Diese Tabelle stellt die Eingabe- und Zielwerte für die neuronalen Netze dar.

Eingabewerte	**Zielwerte**
x_t	x_{t+1}
y_t	y_{t+1}
θ_t	θ_{t+1}
v_t	
ω_t	

Das neuronale Netz soll unter anderem den neuen Ausrichtungswinkel θ_{t+1} des Roboters bestimmen. Dieser Winkel ist in dem Intervall $(-\pi, \pi]$ angegeben, wobei ein Winkel von π dieselbe Ausrichtung wie ein Winkel von $-\pi$ darstellt. Wenn jedoch beispielsweise ein Winkel von $\theta_{t+1} = \pi$ der Zielwert wäre und das neuronale Netz einen Winkel von $\hat{\theta}_{t+1} = -\pi$ bestimmt, würde durch eine Kostenfunktion wie die mittlere quadratische Abweichung der größtmögliche Fehler für diesen Wert ermittelt werden, obwohl der Winkel von dem neuronalen Netz exakt bestimmt wurde. Aufgrund dieser falschen Fehlerberechnung für bestimmte Werte von θ_{t+1} kann diese Variable nicht effektiv erlernt werden. Jeder Winkel θ kann durch das Paar aus dem entsprechenden Sinus $\sin(\theta)$ und Kosinus $\cos(\theta)$ repräsentiert werden. Um die falsche Fehlerberechnung beim Erlernen der Winkel zu umgehen, werden statt θ_{t+1} die beiden Werte $\sin(\theta_{t+1})$ und $\cos(\theta_{t+1})$ als Zielwerte für das neuronale Netz benutzt. Der Ausrichtungswinkel des Roboters kann nach der Lernphase anhand der Netzwerkausgaben $\hat{s}$ für $\sin(\theta_{t+1})$ und $\hat{c}$ für $\cos(\theta_{t+1})$ durch $\hat{\theta}_{t+1} = atan2(\hat{s}, \hat{c})$ wieder rekonstruiert werden.

Die endgültigen Eingabe- und Zielwerte für das neuronalen Netz werden in 3.5 als Tabelle dargestellt. Für das Training und Testen der neuronalen Netze werden Paare aus Eingabewerten und dazugehörigen Zielwerten zufällig in ein Trainings-, Validierungs- und Testset aufgeteilt.

Für das Erlernen des Bewegungsmodells wird ein Feedforward-Netz eingesetzt. Bei einem solchen Netzwerk dienen die Ausgabewerte der Neuronen einer Schicht immer als Eingabewerte für die Neuronen der nächsttieferen Schicht. Ein neuronales Netz besteht typischerweise aus drei Arten von Schichten: einer Eingabeschicht, einer beliebigen Anzahl an versteckten Schichten und einer Ausgabeschicht. Die Größe der Eingabeschicht ist durch die Anzahl an Eingabewerten bestimmt. Da für das Erlernen des Bewegungsmodells fünf Eingabewerte verwendet werden, besteht die Eingabeschicht des dafür verwendeten Netzes aus fünf Einheiten, welche die Eingabewerte an die nächste Schicht weitergeben. Die Größe der Ausgabeschicht ist von der Anzahl der zu bestimmenden Werte abhängig. Da insge-

Tabelle 3.5: Diese Tabelle stellt die endgültigen Eingabe- und Zielwerte für die neuronalen Netze dar. Anstelle von θ_{t+1} werden $\sin(\theta_{t+1})$ und $\cos(\theta_{t+1})$ als Zielwerte eingesetzt.

Eingabewerte	**Zielwerte**
x_t	x_{t+1}
y_t	y_{t+1}
θ_t	$\sin(\theta_{t+1})$
v_t	$\cos(\theta_{t+1})$
ω_t	

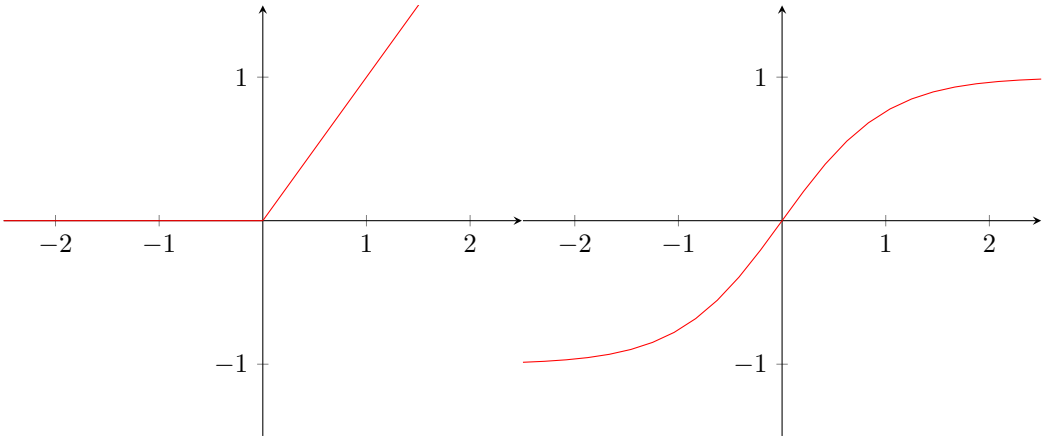
samt vier Werte bestimmt werden sollen, besteht die Ausgabeschicht in diesem Fall aus vier Neuronen.

Das neuronale Netz soll mehrere numerische Werte bestimmen, aus diesem Grund wird für die Neuronen der Ausgabeschicht eine einfache lineare Aktivierungsfunktion gewählt. Im Gegensatz zu der Eingabe- und Ausgabeschicht ist die Anzahl und Größe der versteckten Schichten frei wählbar. Die Aktivierungsfunktion der Neuronen der versteckten Schicht ist außerdem nicht durch die Art der Ausgabewerte des Netzes vorbestimmt. Wie im nächsten Abschnitt beschrieben, wurden in dieser Arbeit Netze mit unterschiedlichem Aufbau der versteckten Schicht getestet. Neuronale Netze mit nur einer versteckten Schicht können jede stetige Funktion approximieren, solange die Aktivierungsfunktion der versteckten Schicht bestimmte Bedingungen erfüllt (Leshno u. a., 1993). Aus diesem Grund wurden Netze mit verschiedenen Größen der einen versteckten Schicht getestet. Als Aktivierungsfunktion wurden für die Neuronen der versteckten Schicht die nicht linearen Funktionen *tanh* und *Rectifier* getestet. *Rectifier* ist eine sehr verbreitete Aktivierungsfunktion für neuronale Netze mit vielen versteckten Schichten. In (Olgac und Karlik, 2011) wurden verschiedene Aktivierungsfunktionen für neuronale Netze mit einer versteckten Schicht getestet, wobei die neuronalen Netze mit *tanh* als Aktivierungsfunktion am besten abschnitten. Wie in Abbildung 3.6 erkennbar ist, bildet die *tanh* Funktion eine Eingabe auf den Bereich $[-1, 1]$ ab und der Ausgabewert der Funktion ist um null zentriert. Die *Rectifier* Funktion setzt alle negativen Eingaben auf Null und gibt alle anderen Eingaben linear wieder.

Im Folgenden wird das Lernverhalten der eingesetzen neuronalen Netze beschrieben. Die Gewichte und Bias-Werte der Neuronen des Netzes werden nach jeder Trainingsepoche aktualisiert. Es wird also durch das gesamte Trainingsset iteriert, bevor die Gewichte und Bias-Werte der Neuronen aktualisiert werden. Als Kostenfunktion wird die mittlere quadratische Abweichung genutzt:

$$e_j(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^{n} (\mathbf{y}_{ij} - f_{ij}(\mathbf{w}, \mathbf{x}))^2 \tag{3.5}$$

wobei $f_{ij}(\mathbf{w}, \mathbf{x})$ der i-te Ausgabewert des Netzes für die j-te Trainingsinstanz in Abhängigkeit von den Gewichten und Bias-Werten $\mathbf{w}$ und den Eingabewerten $\mathbf{x}$ ist. Der i-te Zielwert



(a) Rectifier Aktivierungsfunktion (b) tanh Aktivierungsfunktion

Abbildung 3.6: nichtlineare Aktivierungsfunktionen

für die j-te Trainingsinstanz wird durch y_{ij} dargestellt. Die Kostenfunktion $E(\mathbf{w})$ über eine Epoche mit m Trainingsinstanzen ist dann der Mittelwert der Fehler zu jeder Trainingsinstanz:

$$E(\mathbf{w}) = \frac{1}{m} \sum_{i=1}^{m} e_j(\mathbf{w}) \tag{3.6}$$

Um die Kostenfunktion zu minimieren und nach jeder Trainingsepoche die Gewichte und Bias-Werte des Netzes anzupassen, wird der Levenberg-Marquardt Algorithmus benutzt. Dieser Algorithmus ist sehr effizient für das Trainieren von mittelgroßen Netzwerken mit bis zu mehreren Hundert Gewichten (Hagan und Menhaj, 1994). Außerdem verfügt MATLAB über eine effiziente Implementierung des Algorithmus, sodass dessen Effektivität in einer MATLAB-Umgebung noch deutlicher hervortritt. Der Levenberg-Marquardt Algorithmus ist eine Kombination aus Gradientenverfahren und dem Gauss-Newton Algorithmus.

Bei dem Gradientenverfahren werden, wie in Abschnitt 2.2 beschrieben, die Gewichte angepasst, indem in jeder Iteration des Verfahrens ein Schritt mit einer Schrittweite μ in die entgegengesetzte Richtung des Gradienten der Fehlerfunktion getan wird:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \mu \nabla E\left(\mathbf{w}_k\right) \tag{3.7}$$

wobei $\mathbf{w}_k$ der Vektor der Gewichte und Bias-Werte des Netzes zur Epoche k ist. Ein Nachteil des Gradientenverfahrens ist, dass eine Fehlerfunktion in der Umgebung eines Minimums typischerweise abflacht, wodurch der Gradient in dieser Umgebung kleiner wird. Dadurch kann es dazu kommen, dass das Gradientenverfahren in der Nähe eines Minimums viele Iterationen mit kleinen Schritten braucht, um die Fehlerfunktion zu minimieren.

Bei der Gauss-Newton Fehlerrückführung wird die Fehlerfunktion um $\mathbf{w}_k$ lokal als ein Polynom zweiter Ordnung approximiert. Diese quadratische Approximation der Fehlerfunktion wird minimiert, indem die Gewichte und Bias-Werte wie folgt angepasst werden:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \left[\mathbf{J}^T \mathbf{J}\right]^{-1} \nabla E\left(\mathbf{w}_k\right) \tag{3.8}$$

wobei $\mathbf{J}$ die Jakobimatrix ist, welche die partiellen Ableitungen von allen Fehlertermen $e_j(\mathbf{w})$ nach den einzelnen Gewichten und Bias-Werten w_k beinhaltet. Wenn sich die Fehlerfunktion lokal gut als eine quadratische Funktion approximieren lässt, wie zum Beispiel in der Nähe eines Minimums, wird sie durch die Anpassung der Gewichte und Bias-Werte reduziert. Wenn $\mathbf{w}_k$ jedoch weit von einem Minimum entfernt ist, kann die lokale Approximation der Fehlerfunktion als ein Polynom zweiter Ordnung sehr ungenau sein und die Anpassung der Gewichte und Bias-Werte verringert die Fehlerfunktion nicht zwangsläufig.

Der Levenberg-Marquardt Algorithmus ist wie bereits erwähnt eine Kombination aus dem Gradientenverfahren und dem Gauss-Newton Algorithmus. Das Prinzip des Algorithmus ist es, sich per Gradientenverfahren einem Minimum der Fehlerfunktion zu nähern und dann in der Nähe des Minimums in den Gauss-Newton Algorithmus überzugehen, da dieser in der Umgebung eines Minimums ein besseres Verhalten aufweist. Die Aktualisierungsregel der Gewichte beim Levenberg-Marqaurdt Algorithmus ist folgende:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \left[\mathbf{J}^T \mathbf{J} + \mu \mathbf{I}\right]^{-1} \nabla E\left(\mathbf{w}_k\right) \tag{3.9}$$

wobei μ ein Parameter ist, mit dem beeinflusst wird, ob sich der Algorithmus wie das Gradientenverfahren oder wie den Gauss-Newton Algorithmus verhält.
Für große μ verhält sich der Algorithmus wie das Gradientenverfahren mit einer Lernrate von $1/\mu$ und für kleine μ verhält er sich wie der Gauss-Newton Algorithmus. Der Parameter μ wird nach jeder Iteration des Algorithmus angepasst. Wenn eine Veränderung der Gewichte die Fehlerfunktion reduziert, wird μ um einen β Faktor dividiert. Wenn die Veränderung der Gewichte jedoch nicht zu einer Reduktion der Kostenfunktion führen würde, wird die Veränderung der Gewichte verworfen und μ wird mit dem Faktor β multipliziert. Für dieses neuronale Netz wurde ein Startwert von $\mu = 0.001$ und ein Aktualisierungsfaktor von $\beta = 10$ benutzt.

Um eine Überanpassung des neuronalen Netzes zu verhindern, wird Early Stopping als Regularisierungstechnik angewandt. Wenn das neuronale Netz über mehrere Epochen anhand der Trainingsdaten optimiert wird, kann es passieren, dass das Netz sich zu sehr an den Trainingsdaten anpasst und kein generalisiertes Modell erlernt. In diesem Fall würde das erlernte Modell einen hohen Fehler bei neuen Daten, die nicht im Trainingsset enthalten sind, erzeugen. Um dies zu vermeiden, wird am Ende jeder Trainingsepoche die Genauigkeit des erlernten Modells an den Daten des Validierungssets ermittelt. Die Daten des Validierungssets werden nicht beim Training des Netzes eingesetzt, die Genauigkeit des erlernten Modells wird also anhand neuer Daten überprüft. Wenn sich der Fehler des Modells am Validierungsset im Vergleich zur vorigen Epoche verbessert hat, werden die Gewichte und Bias-Werte dieses Modells gespeichert. Wenn die Genauigkeit des erlernten Modells am Validierungsset über sechs Epochen in Folge abnimmt, wird das Trainieren des Netzes abgebrochen, da dies ein Anzeichen von Überanpassung ist. Wenn dieser Fall eintritt, wird das Modell anhand der Trainingsdaten optimiert, jedoch verschlechtert sich die Genauigkeit des erlernten Modells an neuen, ungesehenen Daten. Nach dem Abbruch des Trainings wird das gespeicherte Modell mit der besten Genauigkeit am Validierungsset zurückgegeben.

Die Genauigkeit des durch das neuronale Netz erlernten Modells ist unter anderem abhängig von den Initialwerten der Gewichte und Bias-Werte des Netzes und von den für

das Training des Netzes ausgewählten Daten. Netze mit unterschiedlichen Initialwerten der Gewichte oder Netze, die auf verschiedenen Trainingsdaten trainiert wurden, können also unterschiedlich genau sein. Um sicher zu stellen, dass ein genaues Modell erlernt wird, wurden 10 Netze mit zufällig initialisierten Gewichten und Bias-Werten und einer zufälligen Aufteilung der Daten in Trainings- und Validierungssets trainiert. Die Genauigkeit von jedem der erlernten Modelle wird an einem Testdatenset ermittelt. Das Modell mit dem niedrigstem Fehler am Testdatenset wird ausgewählt. Hierfür wurden zufällig 15% von den Gesamtdaten als Testdatensatz entnommen. Der Rest der Daten wird von jedem der 10 Netze zufällig in ein Trainings- und Validierungsset eingeteilt, wobei 70% der Daten als Trainingsset und 15% als Validierungsset benutzt werden.

4

Ergebnisse

In diesem Kapitel werden die verschiedenen Bewegungsmodelle evaluiert, welche mittels neuronaler Netze trainiert wurden. Anschließend wird das beste erlernte Bewegungsmodell mit dem in Abschnitt 2.1 vorgestellten analytischen Modell eines Roboters mit Differentialantrieb verglichen. Es wird überprüft, ob das erlernte Bewegungsmodell genauer ist als ein mathematisch aufgestelltes Modell.

Wie im vorherigen Kapitel beschrieben, wurden neuronale Netze mit einer unterschiedlichen Anzahl von Neuronen in der versteckten Schicht getestet. Außerdem wurden die *Rectifier* und *tanh* Funktionen als Aktivierungsfunktion für die Neuronen der versteckten Schicht getestet. Hierfür wurden Netze mit mit einer Größe der versteckten Schicht von 10, 20, 30 und 50 Neuronen trainiert. Für jede dieser Netzwerkarchitekturen wurden 10 Netze mit zufälligen Anfangswerten der Gewichte und Bias-Werte und einer zufälligen Aufteilung von Trainings- und Validierungsdaten trainiert. Dies wurde jeweils für Netzwerke mit der *tanh* und der *Rectifier* Funktion als Aktivierungsfunktion für die Neuronen der versteckten Schicht durchgeführt. Jedes trainierte Netz wurde abschließend an denselben Testdaten evaluiert, welche nicht beim Training der Netzwerke benutzt wurden.

Um die Performanz der Netzwerke mit den beiden verschiedenen Aktivierungsfunktionen und den unterschiedlichen Größen der versteckten Schicht zu vergleichen, wird für jede Netzwerkskonfiguration der Mittelwert des Fehlers der 10 Netzwerke, die mit dieser Konfiguration trainiert wurden, an den Testdaten ermittelt (Tabelle 4.1). Bei jeder getesteten Größe der versteckten Schicht haben die Netze mit *tanh* als Aktivierungsfunktion an den Testdaten besser abgeschnitten als die Netze mit *Rectifier* als Aktivierungsfunktion. Über alle getesteten Größen der versteckten Schicht hinweg gesehen waren die Netze mit *tanh* als Aktivierungsfunktion im Durchschnitt um 1,02% genauer. Von allen Netzwerkskonfigurationen schnitten die neuronalen Netze mit einer versteckten Schicht aus 50 Neuronen, welche *tanh* als Aktivierungsfunktion benutzen, mit einem mittleren Fehler von $6,8031 \times 10^{-4}$ am besten an den Testdaten ab. Das insgesamt beste Netz besteht aus der eben erwähnten Konfiguration und weist einen Fehler von $6,8021 \times 10^{-4}$ an den Testdaten auf.

Das erlernte Modell des neuronalen Netzes mit dem niedrigsten Fehler an den Testdaten wird anschließend mit dem erstellten Differentialantriebsmodell verglichen. Es wird untersucht, wie genau die beiden Modelle anhand der Eingabewerte der Testdaten die Position

Tabelle 4.1: Diese Tabelle stellt die durchschnittlichen Fehler der verschiedenen Netzwerkskonfigurationen an den Testdaten dar.

Anzahl Neuronen	tanh	Rectifier
10	$7,0652 \times 10^{-4}$	$7,1520 \times 10^{-4}$
20	$6,9476 \times 10^{-4}$	$7,0144 \times 10^{-4}$
30	$6,9008 \times 10^{-4}$	$6,9464 \times 10^{-4}$
40	$6,8268 \times 10^{-4}$	$6,9065 \times 10^{-4}$
50	$6,8031 \times 10^{-4}$	$6,8798 \times 10^{-4}$
Mittelwert	$6,9087 \times 10^{-4}$	$6,9798 \times 10^{-4}$

$[x_k, y_k]^T$ und den Ausrichtungswinkel θ_k der Zielwerte der Testdaten bestimmen. Die Genauigkeit der Modelle wird anhand der mittleren absoluten Abweichung der Ausgabewerte von den Zielwerten der Testdaten bestimmt. Der Fehler bei der Bestimmung der Position wird wie folgt ermittelt:

$$\varepsilon_{Position} = \frac{1}{n} \sum_{k=1}^{n} \left\| \begin{bmatrix} x_k - \hat{x}_k \\ y_k - \hat{y}_k \end{bmatrix} \right\| \tag{4.1}$$

wobei x_k und y_k die Zielwerte, $\hat{x}_k$ und $\hat{y}_k$ die Ausgaben des jeweiligen Modells und n die Anzahl der Testdaten sind. Der Fehler bei der Bestimmung des Winkels wird wie folgt berechnet:

$$\varepsilon_{Winkel} = \frac{1}{n} \sum_{k=1}^{n} \left| angdiff\left(\hat{\theta}_k,\, \theta_k\right) \right| \tag{4.2}$$

wobei $angdiff\,(\alpha, \beta)$ eine Funktion ist, welche α von β subtrahiert und das Ergebnis dann auf dem Intervall $(-\pi, \pi]$ angibt. Das durch das neuronale Netz erlernte Modell weist bei der Positionsbestimmung einen Fehler von $2,89 \times 10^{-2}$ auf und bei der Winkelbestimmung einen Fehler von $3,21 \times 10^{-2}$. Die mittleren Fehler des erlernten Modells werden in Abbildung 4.2 als Histogramme dargestellt. Das Differentialantriebsmodell bestimmt die Position mit einem Fehler von $6,69 \times 10^{-2}$ und den Winkel mit einem Fehler von $3,72 \times 10^{-2}$. Die mittleren Fehler des Differentialantriebsmodell werden in 4.3 dargestellt. Das erlernte Modell ist also bei der Positionsbestimmung um 56,79% genauer und bei der Winkelbestimmung um 13,74% genauer.

Um die Genauigkeit der erlernten Bewegungsmodelle weiter zu verbessern, wurden weitere neuronale Netze mit zusätzlichen Eingabewerten trainiert. Die bisher trainierten neuronalen Netze benutzten dieselben Eingabewerte wie das analytisch aufgestellte Bewegungsmodell: die Positionswerte $\{x_i\}$ und $\{y_i\}$, den Ausrichtungswinkel $\{\theta_i\}$ und die Geschwindigkeitsbefehle $\{v_i\}$ und $\{\omega_i\}$ zu den Zeitpunkten $i = 1, 2, ..., n-1$. Nun werden die Veränderungen dieser Eingabewerte über den vorherigen Zeitschritt als zusätzliche Eingaben eingesetzt. Diese zusätzlichen Eingabewerte Δx, Δy, $\Delta\theta$, Δv und $\Delta\omega$ sind wie folgt

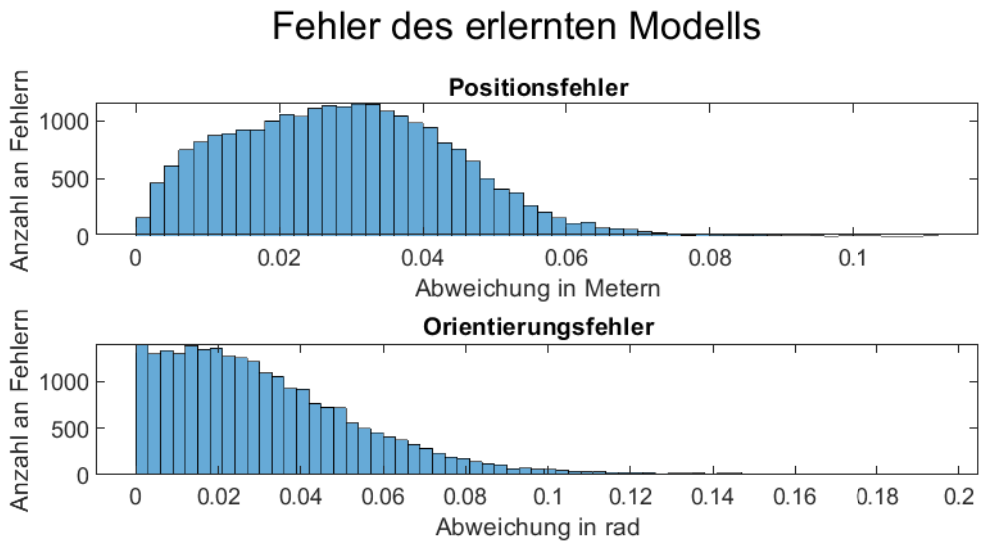


Abbildung 4.2: Diese Abbildung stellt die Fehler des durch das neuronale Netz erlernten Modells bei der Bestimmung der Position und der Orientierung als Histogramm dar.

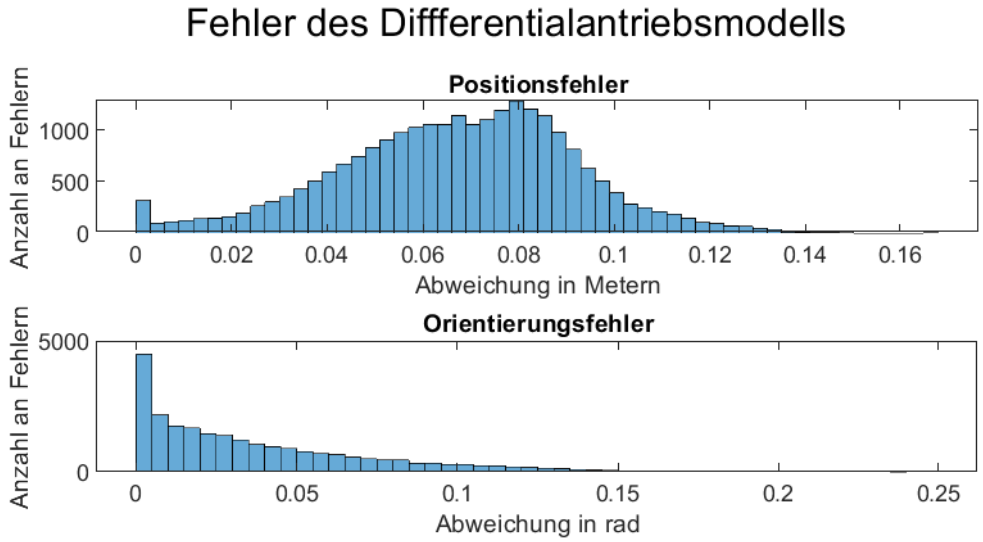


Abbildung 4.3: Diese Abbildung stellt die Fehler des analytisch aufgestellten Differentialantriebsmodells bei der Bestimmung der Positions und der Orientierung als Histogramm dar.

definiert:

$$\begin{aligned}
\Delta x &= \{x_1, x_2 - x_1, x_3 - x_2, ..., x_{n-1} - x_{n-2}\} \\
\Delta y &= \{y_1, y_2 - y_1, y_3 - y_2, ..., y_{n-1} - y_{n-2}\} \\
\Delta \theta &= \{\theta_1, angdiff\left(\theta_1, \theta_2\right), angdiff\left(\theta_2, \theta_3\right), ..., angdiff\left(\theta_{n-2}, \theta_{n-1}\right)\} \\
\Delta v &= \{v_1, v_2 - v_1, v_3 - v_2, ..., v_{n-1} - v_{n-2}\} \\
\Delta \omega &= \{\omega_1, \omega_2 - \omega_1, \omega_3 - \omega_2, ..., \omega_{n-1} - \omega_{n-2}\}
\end{aligned} \tag{4.3}$$

Diese neuronalen Netze mit zusätzlichen Eingabewerten wurden nach der bereits erwähnten Vorgehensweise mit verschiedenen Netzwerkkonfigurationen trainiert und dann ausgewertet. Die mittleren Fehler der neuen Netze mit den verschiedenen Netzwerkkonfigurationen sind in Tabelle 4.4 dargestellt. Wie bei den bisher trainierten Netzen sind die neuronalen Netze mit *tanh* als Aktivierungsfunktion für jede getestete Größe der versteckten Schicht genauer als die Netze, welche *Rectifier* benutzen. Über alle Größen der versteckten Schicht hinweg sind die Netzwerke mit *tanh* als Aktivierungsfunktion um 4,56% genauer als die Netze mit *Rectifier*. Die Netzwerkkonfiguration der neuen neuronalen Netze mit dem niedrigsten mittleren Fehler an den Testdaten sind die Netze mit einer versteckten Schicht bestehend aus 40 Neuronen, welche *tanh* als Aktivierungsfunktion benutzen. Das beste getestete Netzwerk besteht aus dieser Architektur und weist einen Fehler von $3,5510 \times 10^{-4}$ auf.

Die neuronalen Netze mit zusätzlichen Eingabewerten weisen bei jeder getesteten Netzwerkkonfiguration einen niedrigeren mittleren Fehler bei den Testdaten auf als die bisher getesteten Netzwerke mit weniger Eingabewerten. Über alle getesteten Größen der versteckten Schicht hinweg ist der mittlere Fehler der neuen Netze mit *tanh* als Aktivierungsfunktion um 46,35% niedrigerer als der Fehler der alten Netze. Mit *Rectifier* als Aktivierungsfunktion sind die Netze mit zusätzlichen Eingabewerten um 44,36% genauer als die Netze mit weniger Eingabewerten. Das beste getestete neuronale Netz mit zusätzlichen Eingabewerten ist an den Testdaten um 47,80% genauer als das beste neuronale Netz mit weniger Eingabewerten.

Tabelle 4.4: Diese Tabelle stellt die durchschnittlichen Fehler der verschiedenen Netzwerkskonfigurationen der neuronalen Netze mit zusätzlichen Eingabewerten an den Testdaten dar.

Anzahl Neuronen	**tanh**	**Rectifier**
10	$3,7310 \times 10^{-4}$	$3,9557 \times 10^{-4}$
20	$3,8541 \times 10^{-4}$	$4,0092 \times 10^{-4}$
30	$3,6312 \times 10^{-4}$	$3,9596 \times 10^{-4}$
40	$3,6114 \times 10^{-4}$	$3,6710 \times 10^{-4}$
50	$3,7047 \times 10^{-4}$	$3,8216 \times 10^{-4}$
Mittelwert	$3,7065 \times 10^{-4}$	$3,8834 \times 10^{-4}$

Das durch das beste neuronale Netz mit zusätzlichen Eingabewerten erlernte Modell wird nun mit dem analytisch aufgestellten Differentialantriebsmodell verglichen. Hierfür werden wieder die in 4.1 und 4.2 definierten Fehler für die Bestimmung der Position und

Orientierung verwendet. Das neue erlernte Modell weist bei der Bestimmung der Position einen Fehler von $1,20 \times 10^{-2}$ auf und bei der Bestimmung des Winkels einen Fehler von $2,42 \times 10^{-2}$. Damit ist der Positionsfehler des neuen erlernten Modells um 82,07% geringer als der Positionsfehler des Differentialantriebsmodells und der Orientierungsfehler des neuen erlernten Modells ist um 34,84% geringer. Das neue erlernte Modell reduziert den Fehler bei der Bestimmung der Position im Vergleich zu dem alten erlernten Modell um 59,48% und den Fehler bei der Bestimmung des Winkels um 24,61%.

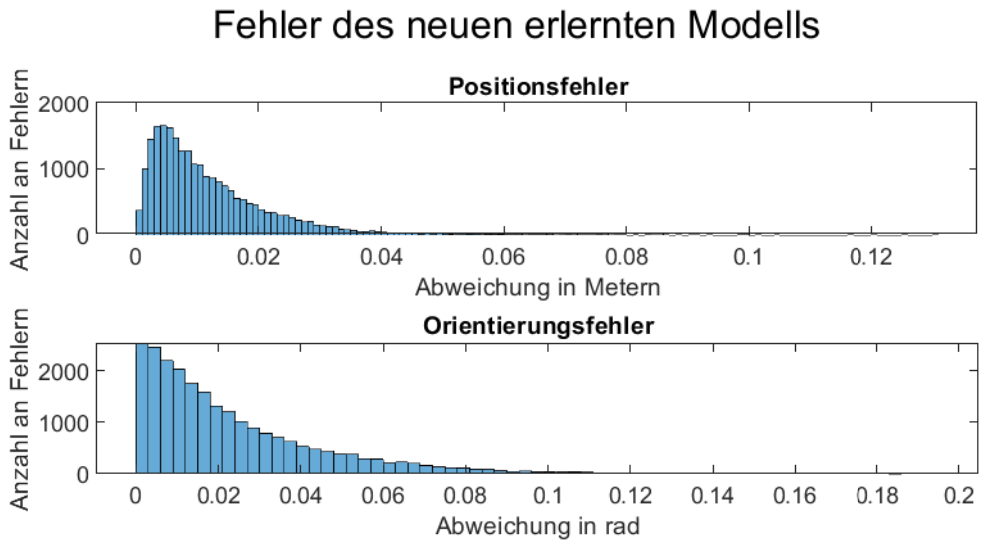


Abbildung 4.5: Diese Abbildung stellt die Fehler des durch das neuronale Netz mit zusätzlichen Eingabewerten erlernten Modells bei der Bestimmung der Position und der Orientierung als Histogramm dar.

5

Zusammenfassung und Ausblick

Das Ziel dieser Arbeit war es, ein Bewegungsmodell für den *Loomo Segway* Roboter zu erlernen, das dafür eingesetzt werden kann, die bereits implementierte lokale Pfadplanung für diesen Roboter zu verbessern. Um die hierfür benötigten Daten zu generieren, wurde zunächst eine Datenaufnahme durchgeführt, bei der die Bewegung des Segwayroboters durch ein Trackingsystem und der Odometrie des Roboters erfasst wurde. Für das Erlernen des Bewegungsmodells wurden verschiedene künstliche neuronale Netze getestet. Es wurden Feedforward-Netze mit einer versteckten Schicht eingesetzt, wobei Netze mit 10, 20, 30, 40 und 50 Neuronen in der versteckten Schicht getestet wurden. Es wurden außerdem für jede Größe der versteckten Schicht die nichtlinearen Aktivierungsfunktionen *tanh* und *Rectifier* getestet. Des Weiteren wurde untersucht, ob die Performanz der neuronalen Netze gesteigert werden kann, indem zusätzliche Eingabewerte genutzt werden. Das beste erlernte Modell wurde validiert, indem dessen Genauigkeit mit der eines analytisch aufgestellten Bewegungsmodells an den aufgenommenen Bewegungsdaten verglichen wurde. Es wurde gezeigt, dass das erlernte Bewegungsmodell die Position und Orientierung des *Loomo Segway* Roboters zum nächsten Zeitpunkt besser bestimmt als das analytisch aufgestellte Modell.

Bei der Evaluation der erlernten Modelle an den aufgenommenen Bewegungsdaten wurde festgestellt, dass über alle getesteten Größen der versteckten Schicht hinweg die neuronalen Netze mit *tanh* als Aktivierungsfunktion für die Neuronen der versteckten Schicht besser abschnitten als die Netze, welche *Rectifier* als Aktivierungsfunktion nutzen. Außerdem wurde festgestellt, dass die Performanz der neuronalen Netze durch das Hinzufügen zusätzlicher Eingabewerte deutlich gesteigert wurde und dass das Hinzufügen zusätzlicher Eingabewerte einen weitaus größeren Einfluss auf die Leistung der neuronalen Netze hatte als die Wahl der Aktivierungsfunktion oder die Größe der versteckten Schicht. Aufgrund dieser scheinbar großen Bedeutung von zusätzlichen Eingabewerten für die Leistung der neuronalen Netze wäre es interessant, bei einer zukünftigen Datenaufnahme weitere Eigenschaften des *Loomo Segway* Roboters, wie dessen Neigungswinkel, aufzuzeichnen. Unter Berücksichtigung dieser zusätzlichen Eigenschaften könnten die erlernten Modelle noch weiter verbessert werden. Außerdem wäre es interessant, nicht nur neuronale Netze mit verschiedenen Größen der einen versteckten Schicht zu untersuchen, sondern auch Netze mit zusätzlichen versteckten Schichten. Durch zusätzliche versteckte Schichten könnte die Leistung der eingesetzten neuronalen Netze weiter gesteigert werden.

In dieser Arbeit wurde gezeigt, dass ein Bewegungsmodell für den *Loomo Segway* Roboter erlernt werden kann, welches die Bewegung des Roboters besser modelliert als ein klassisches analytisch aufgestelltes Bewegungsmodell für einen Roboter mit Differentialantrieb. Ein solches erlerntes Bewegungsmodell kann also dafür eingesetzt werden, die lokale Pfadplanung für den *Loomo Segway* Roboter zu verbessern.

Literatur

Arun, K.S., Huang, T.S. und Blostein, S.D. (1987). Least-Squares Fitting of Two 3-D Point Sets. IEEE Transactions on Pattern Analysis and Machine Intelligence *PAMI-9*, 698–700. DOI: 10.1109/TPAMI.1987.4767965.

Eliazar, A.I. und Parr, R. (2004). Learning Probabilistic Motion Models for Mobile Robots. In Proceedings of the Twenty-First International Conference on Machine Learning, ICML '04. Banff, Alberta, Canada: Association for Computing Machinery, S. 32. ISBN: 1581138385. DOI: 10.1145/1015330.1015413.

Garrote, L., Temporão, D., Temporão, S., Pereira, R., Barros, T. und Nunes, U.J. (2020). Improving Local Motion Planning with a Reinforcement Learning Approach. In 2020 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC), S. 206–213. DOI: 10.1109/ICARSC49921.2020.9096095.

Gu, D. und Hu, H. (Mai 2002). Neural Predictive Control for a Car-like Mobile Robot. Robotics and Autonomous Systems *39*, 73–86. DOI: 10.1016/S0921-8890(02)00172-0.

Hagan, M.T. und Menhaj, M.B. (1994). Training feedforward networks with the Marquardt algorithm. IEEE Transactions on Neural Networks *5*, 989–993. DOI: 10.1109/72.329697.

Leshno, M., Lin, V.Y., Pinkus, A. und Schocken, S. (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. Neural Networks *6*, 861–867. ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(05)80131-5.

NaturalPoint Inc. (2021). *General FAQs*. Abgerufen am 03. Februar 2021. URL: https://www.optitrack.com/support/faq/general.html.

Olgac, A und Karlik, B. (Feb. 2011). Performance Analysis of Various Activation Functions in Generalized MLP Architectures of Neural Networks. International Journal of Artificial Intelligence And Expert Systems *1*, 111–122.

SegwayRobotics Inc. (2019). *robot reference frame*. Abgerufen am 27. Dezember 2020. URL: http://developer.segwayrobotics.com/images/developer/robot-reference-frame.jpg.

Thrun, S., Burgard, W. und Fox, D. (2005). Probabilistic Robotics (Intelligent Robotics and Autonomous Agents). The MIT Press. ISBN: 0262201623.

Wehbe, B., Hildebrandt, M. und Kirchner, F. (2017). Experimental evaluation of various machine learning regression methods for model identification of autonomous underwater

vehicles. In 2017 IEEE International Conference on Robotics and Automation (ICRA), S. 4885–4890. DOI: 10.1109/ICRA.2017.7989565.