Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Sebastian Krome

Einparken eines autonomen Fahrzeugs durch überwachtes Lernen

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Sebastian Krome

Einparken eines autonomen Fahrzeugs durch überwachtes Lernen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Ing. Andreas Meisel
Zweitgutachter : Prof. Dr. rer. nat. Reinhard Baran

Abgegeben am 1. Oktober 2013

Sebastian Krome

Thema der Arbeit
Einparken eines autonomen Fahrzeugs durch überwachtes Lernen

Stichworte
Maschinelles Lernen, künstliche Neuronale Netze, künstliche modulare Neuronale Netze, autonomes Einparken

Kurzzusammenfassung
In dieser Arbeit wurden in der Simulation unterschiedliche Verfahren entwickelt und verglichen, wie ein autonomes Fahrzeug durch Methoden des überwachten Lernens das Einparken in eine Parklücke erlernen kann. In diesem Kontext wurden verschiedene Möglichkeiten der Zustandsrepräsentation, sowie der Einsatz von künstlichen Neuronalen Netzen und modularen künstlichen Neuronalen Netzen zur Funktionsapproximation untersucht.
Sebastian Krome

Title of the paper
Autonomous car parking via supervised learning

Keywords
Machine Learning, Artificial Neural Networks, Modular Artificial Neural Networks, Autonomous Car Parking

Abstract
In this work, different methods for parking an autonomous vehicle by methods of supervised learning have been developed and compared. In this context different possibilities of state representation, and the use of artificial neural networks and modular artificial neural networks for function approximartion have been investigated.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Fahrerassistenzsysteme gehören immer öfter zur Ausstattung eines Autos und es gibt eine Vielzahl von ihnen. Sie alle haben zum Ziel, Unfälle zu verhindern und das Fahren einfacher und komfortabler zu machen [e.V13].

Dabei nehmen sie uns verschiedene Aufgaben ab. Sie helfen beim Spurwechsel, indem sie während des Vorgangs die Fahrzeugumgebung beobachten und insbesondere vor anderen Fahrzeugen im toten Winkel warnen. Sie erkennen drohende Auffahrunfälle und bereiten die Bremsen auf ein schnelleres Bremsen vor oder bremsen selbstständig. Sie regeln das Ein- und Ausschalten des Fernlichts, erkennen Verkehrszeichen und helfen dem Fahrer dabei, das Fahrzeug einzuparken, was Thema dieser Arbeit ist.

Ein Einparkassistent unterstützt den Fahrer dabei, eine passende Parklücke zu finden und das Fahrzeug schnell und sicher einzuparken. Eine Schwierigkeit dabei ist das Abschätzen der Fahrzeuggeometrie im Heck- und Frontbereich, sowie designbedingte Gestaltungen, wie Säulen und Fensterflächen, die die Übersichtlichkeit während des Einparkprozesses einschränken können [WHW12].

Es gibt zahlreiche verschiedene Varianten von Einparkassistenten, die sich in folgende Kategorien einteilen lassen[Alp10] :

Informierende Informationssysteme: Hierzu gehören Systeme, die die Entfernungen zu Objekten messen, die sich in der Nähe des Fahrzeug befinden und den Fahrer über diese informieren. Ebenso solche, die der Parklückenvermessung dienen und so bei der Auswahl einer geeigneten Parklücke helfen.

Geführte Einparkassistenz: Sie bewerten die Umfeldinformationen und geben dem Fahrer konkrete Handlungsempfehlungen.

Semiautomatisches Einparken: Hierbei wird dem Fahrer eine Fahrzeugführungskomponente vom System abgenommen. Üblich sind Systeme, bei denen das System für die Lenkung sorgt, während der Fahrer die Geschwindigkeitssteuerung übernimmt.

Vollautomatisches Einparken: Die gesamte Fahrzeugführung wird vom System übernommen.

1.1 Ziele

Das Ziel dieser Arbeit ist es, ein Verfahren zu entwickeln, wie ein autonomes Fahrzeug mit Hilfe von Methoden des überwachten Lernens das Einparken in eine Parklücke erlernen kann. Hierzu müssen aus Lerndaten, die während des Trainings durch einen menschlichen Trainer gewonnen werden, konkrete Aktionen hergeleitet (erlernt) werden, durch die das Fahrzeug nach dem Training selbstständig in eine Parklücke navigieren kann.

1.2 Gliederung der Arbeit

Die Arbeit ist in mehrere Teile aufgeteilt. In Abschnitt 2 werden zunächst exemplarisch unterschiedliche Ansätze zur Lösung des Einparkproblems durch Lernverfahren vorgestellt.
In Abschnitt 3 werden die Grundlagen der verschiedenen in dieser Arbeit genutzten Verfahren zur Funktionsapproximation und zum Aufteilen von Daten in Cluster vorgestellt.
Im Anschluss wird in Abschnitt 4 die konkrete Umsetzung der einzelnen Anforderungen beschrieben.
Im letzten Abschnitt (Abschnitt 5) werden dann die einzelnen entwickelten Szenarien zusammen mit den Ergebnissen der Einparkversuche vorgestellt.

2 Ansätze zum automatischen Einparken

In diesem Abschnitt wird ein kurzer Einblick in andere Arbeiten gegeben, in denen verschiedene Einparkassistenten entwickelt wurden.

2.1 Automatisches Einparken durch Speichern von erfolgreichen Zustand-Aktions Paaren

In der hier vorgestellten Arbeit bestand das Ziel darin, ein Fahrzeug selbstständig lernen zu lassen, in eine Parklücke einzuparken. Hierfür standen Sensorinformationen zur Verfügung, zu denen ein passender Lenkwinkel gelernt werden sollte [NYWI08].

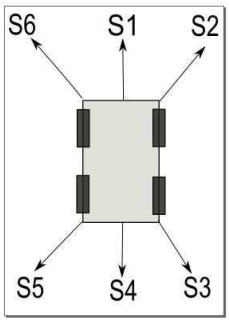


Abbildung 2.1: Ausrichtung der Entfernungssensoren. Angelehnt an [NYWI08]

In Abbildung 2.1 wird ein Modell des dazu verwendeten Fahrzeugs gezeigt, aus der ebenfalls die Ausrichtung der 6 verwendeten Distanzsensoren hervorgeht.

Die gemessenen Entfernungen werden zunächst in 8 diskrete Werte eingeteilt, um auf diese Weise den aktuellen Zustand des Fahrzeugs zu beschreiben. Aus dem aktuellen Zustand sollte im nächsten Schritt eine geeigneter Lenkwinkel erlernt werden. Dieser wurde in die diskreten Werte $\pm\pi/8;\ \pm\pi/4;\ 0$ eingeteilt. Die Geschwindigkeit ist konstant.

Zum Erlernen des Lenkwinkels wurde die RPM-Methode verwendet (rational policy making

S1	S2	S3	S4	S5	S6	Aktion (Lenkwinkel)
0,8	1	3	3	3	0,5	π/8
1,2	0,3	3	3	2,7	0,3	π/4
0,7	0,5	1,5	1,4	2,4	1,1	π

Abbildung 2.2: Beispiel für eine mögliche Zustand-Aktions Tabelle

method). Hierbei besitzt der Agent einen primären und einen sekundären Speicher. In diesen sind die Aktionen gespeichert, die der Agent in einem bestimmten Zustand ausgeführt hat.

Während des Lernvorgangs bestimmt der Agent seinen aktuellen Zustand, welcher durch die von den Sensoren gemessenen Entfernungen gegeben ist. Zu diesem Zustand wird nun eine Aktion gewählt. Ist für den aktuellen Zustand bereits eine Aktion im sekundären Speicher abgelegt, wird der Agent diese Aktion wählen. Wenn hingegen noch keine Aktion bekannt ist, wird der Agent eine zufällige Aktion wählen und diese im primären Speicher ablegen. Wie ein Ausschnitt dieser Zustand-Aktions Tabellen aussehen könnte ist beispielhaft in Abbildung 2.2 gezeigt (mit fiktiven Werten). Der Zustand wird durch die diskretisierten Werte der Entfernungssensoren beschrieben. Mit diesen ist eine konkrete Aktion in Form eines Lenkwinkels verbunden.

Sollte der Einparkvorgang korrekt beendet werden, werden alle Aktionen, die im primären Speicher gespeichert wurden, in den sekundären kopiert. Wird der Einparkvorgang nicht korrekt beendet, wird der primäre Speicher gelöscht, ebenso die für diesen nicht erfolgreichen Einparkvorgang genutzten Aktionen aus dem sekundären Speicher.
Zwei Beispiele für Einparkvorgänge, die auf diese Weise erlernt wurden, sind im Abbildung 2.3 zu sehen.

2.2 Approximation von Parametern zur Fahrzeugsteuerung

F. Osório, F. Heinen und L. Fortes haben mit SEVA2D ebenfalls ein System entwickelt, welches Methoden des überwachten Lernens zum Einparken eines Fahrzeugs benutzt. [OHF02]. Dazu wurde eine Simulationsumgebung entwickelt, in der ein Fahrzeug in eine Parklücke einparken sollte. Hierfür wurde zunächst ein endlicher Automat entwickelt, der für die Fahrzeugsteuerung zuständig war.

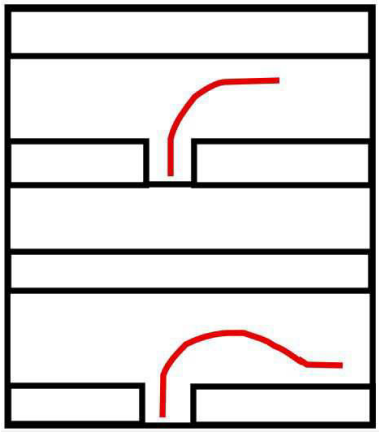


Abbildung 2.3: Zwei erlernte Trjeketorien mit verschiedenen Startpositionen. Angelehnt an [NYWI08]

Der Automat kannte verschiedene Einparkphasen (z.B. Suchen einer Parklücke, Positionierung, Ausrichten). Die Einparkphasen waren mit konkreten, hart programmierten Aktionen verbunden. Auch die Übergänge von einer Einparkphase zu der darauf folgenden waren fest programmiert. Zur Orientierung dienten Entfernungssensoren.

In einem nächsten Schritt wurde der entwickelte Automat durch ein künstliches Neuronales Netz (Jordan Cascade Correlation Netz) nachgebaut. Durch das Neuronale Netz wird eine Approximationsfunktion gefunden, die die von den Entfernungssensoren gemessenen Entfernungen und die aktuelle Einparkphase auf eine Soll-Geschwindigkeit, einen Soll-Lenkwinkel und eine neue Einparkphase abbildet. Dies ist in Abbildung 2.6 zu sehen.

Um verschiedene Schwächen der Simulationsumgebung zu eliminieren und das Projekt weiterzuentwickeln, wurde in einem nächsten Schritt SEVA3D entwickelt [HOHK06]. Durch die neue Simulationsumgebung konnte die Simulation hier nun im dreidimensionalen Raum ausgeführt werden. Weiterhin war es möglich, den simulierten Sensordaten zufälliges Rauschen beizumischen.

Wie bei SEVA2D wurde auch hier zunächst ein endlicher Automat entwickelt. Die Regeln für die Übergänge von einer Einparkphase zur nächsten (siehe Abbildung 2.4) und die Regeln, welche das Verhalten des Fahrzeugs in einer bestimmten Situation bestimmten, waren manuell

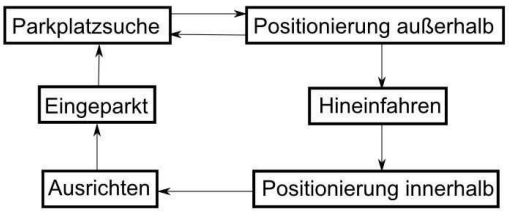


Abbildung 2.4: Einparkphasen und Übergänge bei SEVA3d. Angelehnt an [HOHK06]

entworfen. Die Einparkphasen entsprechen ebenfalls jenen aus SEVA2D. Zur Orientierung in der Umwelt wurden auch wieder Entfernungssensoren genutzt, deren Anordnung in Abbildung 2.5 zu sehen ist.

In einem nächsten Schritt wurden dann mit diesem System Daten generiert, die für das Training

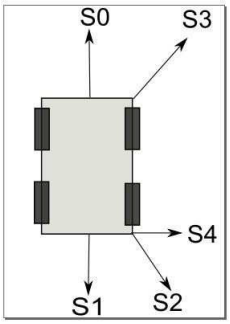


Abbildung 2.5: Anordnung der Sensoren bei SEVA3D. Angelehnt an [HOHK06]

eines Neuronalen Netzes (ebenfalls ein Jordan Netz), das den endlichen Automaten ersetzen sollte, benutzt wurden. Die Eingaben in das Netz waren:

Die aktuelle Einparkphase:

- Suche nach einem Parkplatz (Parkplatzsuche)
- Positionierung außerhalb der Parklücke (Positionierung außerhalb)
- Hineinfahren in die Parklücke (Hineinfahren)
- Positionierung in der Parklücke (Positionierung innerhalb)
- Ausrichten (Ausrichten)
- Eingeparkt (Eingeparkt)

Sensorinformationen: Die Werte der 5 Sensoren. Die Anordnung der Sensoren ist in 2.5 gezeigt.

Aus den Eingaben berechnet des Neuronale Netz:

Geschwindigkeit: 3 Werte: vorwärts, rückwärts und stehend.

Lenkwinkel: 3 Werte: gerade, rechts voll eingeschlagen, links voll eingeschlagen.

Nächste Einparkphase: Entweder identisch mit der aktuellen oder Änderung zu einer auf die aktuelle Phase folgenden Einparkphase. Die aktuelle Phase spielt eine wichtige Rolle dabei, die Sensordaten korrekt auszuwerten. Bei der Zustandsrepräsentation durch Sensordaten kann es vorkommen, dass in der gleichen Situation (das Fahrzeug befindet sich an der selben Position) abhängig von der aktuellen Einparkphase unterschiedliche Aktionen nötig sind („*same situation, different actions* “ [HOHK06]).

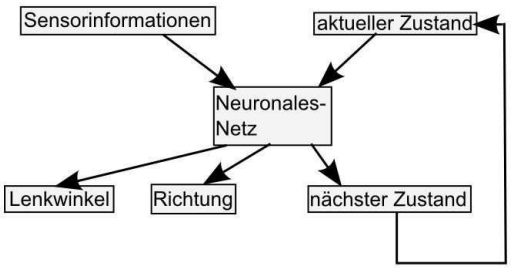


Abbildung 2.6: Eingaben und Ausgaben des Neuronalen Netzes. Angelehnt an [HOHK06]

2.3 Führung des Fahrzeugs entlang einer gelernten Trajektorie

Samani, Ghaisari und Danesh [SGD12a] haben sich mit der Problematik befasst, wie ein lernender Agent mit Hilfe von drei Beispieltrajektorien das Einparken in eine Parklücke erlernen kann. Die Beispieltrajektorien unterscheiden sich dadurch voneinander, dass die y-Koordinate, an der das Fahrzeug den Einparkvorgang startet, unterschiedlich gewählt ist (siehe Abbildung 2.7). Ziel ist es nun, dass der Agent für eine nicht trainierte Startposition zwischen diesen Beispieltrajektorien interpolieren kann.

Hierzu wurden die Beispieltrajektorien in jeweils 486 Punkte eingeteilt. Das Fahrzeug selber

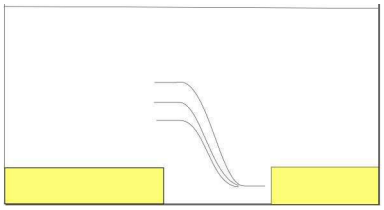


Abbildung 2.7: Beispieltrajektorien. Angelehnt an [SGD12a]

verfügt an seiner linken, vorderen Ecke über zwei Ultraschallsensoren (S_1 und S_2). Diese messen die Entfernungen zu dem Fahrzeug, welches die Parklücke zur einen Seite begrenzt. Zu jedem der Beispielpunkte wurden die gemessenen Entfernungen der Ultraschallsensoren ermittelt.

Mit Hilfe dieser Lerndaten wurde ein Radiale Basis Funktionen Netz trainiert, welches mit Hilfe der drei Beispieltrajektorien weitere Trajektorien zwischen den Beispieltrajektorien interpolieren sollte. Die Eingaben an dieses Netzwerk waren:

1. Die x-Koordinate des aktuellen Punkts.
2. Die von den Sensoren gemessenen Entfernungen (d_1 und d_2).
3. 2 weitere Werte, die auf den von den Sensoren gemessenen Entfernungen basieren (d_3 und d_4).

Abbildung 2.8: Approximation der korrespondierenden y-Koordinate

Diese Eingaben werden auf eine y-Koordinate abgebildet (siehe Abbildung 2.8) . Auf diese Weise lernt der Agent, eine Trajektorie zu generieren.

Um dieser Trajektorie zu folgen, wurde neben der Approximartionskomponente, welche für Generierung der interpolierten Trajektorie zuständig ist, Regelungstechnik verwendet, um der erzeugten Trajektorie zu folgen.

3 Theoretische Grundlagen

3.1 Maschinelles Lernen

Verfahren, die maschinelles Lernen anwenden, sind so entworfen, dass sie anhand von Daten Wissen erlernen können. Dabei wird ein bestimmtes Leistungskriterium anhand von Beispieldaten oder Erfahrungswerten aus der Vergangenheit optimiert.

Hierzu wird ein bestimmtes Modell definiert. Das Lernen besteht nun darin, die Parameter dieses Modells mit Hilfe von Trainingsdaten insofern zu optimieren, dass dieses Modell nach dem Training in der Lage ist, für diese gelernten Daten ein gewünschtes Ergebnis zu erzeugen .

Weiterhin ist es wünschenswert, dass dieses Modell nach dem Training in der Lage ist, ausreichend zu generalisieren. Es soll also nicht nur für die gelernten Daten korrekte Ergebnisse liefern, sondern auch aus den gelernten Daten korrekte Ergebnisse für ähnliche Eingaben ableiten können.

Aufgrund unterschiedlicher Lernprozesse kann man Lernverfahren in verschiedene Kategorien einteilen: das überwachte, das unüberwachte und das bestärkende Lernen.

Überwachtes Lernen bedeutet, dass das System aus Trainingsdaten, bei denen sowohl die Eingaben X als auch die Ausgaben Y bekannt sind, lernt. Die Aufgabe besteht nun darin, die Abbildung von der Eingabe auf die Ausgabe zu lernen. Dies wird in Abbildung 3.1 dargestellt.

Abbildung 3.2 zeigt die Lernsituation „Unüberwachtes Lernen“. Beim unüberwachten Lernen besteht die Aufgabe ebenfalls darin eine Abbildung von Eingabe- auf Ausgabewerte zu erlernen. Allerdings sind hier die Ausgabewerte nicht bekannt und es stehen nur die Eingaben zur Verfügung. Das Ziel besteht nun darin, Regelmäßigkeiten in den Eingaben zu erkennen, sodass man diese in geeignete Klassen einteilen oder mögliche Assoziationen finden kann.

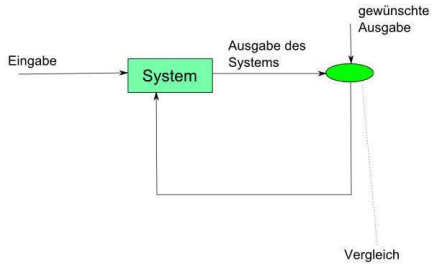


Abbildung 3.1: Überwachtes Lernen

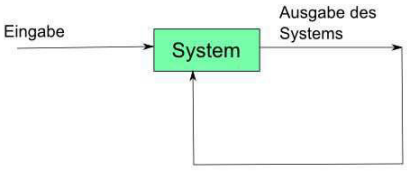


Abbildung 3.2: Unüberwachtes Lernen

Beim bestärkenden Lernen, ist das lernende System ein entscheidungstreffender Agent. Er führt Aktionen aus und ändert auf diese Weise die Umwelt. Gelangt er auf diese Weise in eine neue Umwelt, wird diese durch einen Zustand beschrieben. Dieser gibt den momentanen Zustand der Umwelt inklusive dem Agenten an. Nachdem der Agent eine Aktion ausgeführt hat, bekommt er für diese eine Belohnung oder Bestrafung, abhängig davon ob seine Aktion positiv oder negativ war. Auf diese Weise erlernt er nach einer Reihe von Versuchsdurchläufen ein bestimmtes, gewünschtes Verhalten. Abbildung 3.3 veranschaulicht dieses Vorgehen.

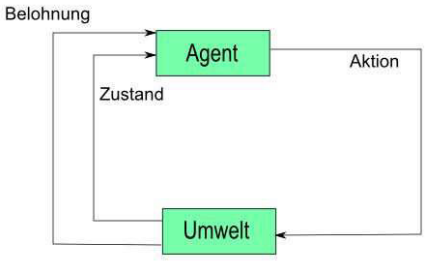


Abbildung 3.3: Bestärkendes Lernen

Tiefer gehende Informationen zu den zuvor dargestellten Sachverhalte sind in [Alp08] zu finden.

3.2 Neuronale Netze

Neuronale Netze sind parallele Verarbeitungseinheiten. Sie bestehen aus einer Vielzahl von miteinander vernetzen einfacher Verarbeitungseinheiten (Neuronen).

3.2.1 Aufbau eines Neurons

Einzelne Neuronen erhalten über gerichtete Verbindungen Eingangssignale, welche verarbeitet werden und ein Ausgangssignal erzeugen. Die Verarbeitungsmechanismen eines Neurons alleine sind dabei in der Regel stark eingeschränkt.

Mit jedem Eingangssignal $x_i \in \mathbb{R}$ ist ein Verbindungsgewicht $w_{i,j} \in \mathbb{R}$ assoziiert.

Abbildung 3.4 zeigt das Schema eines Neurons. Die Eingangssignale werden über die ge-

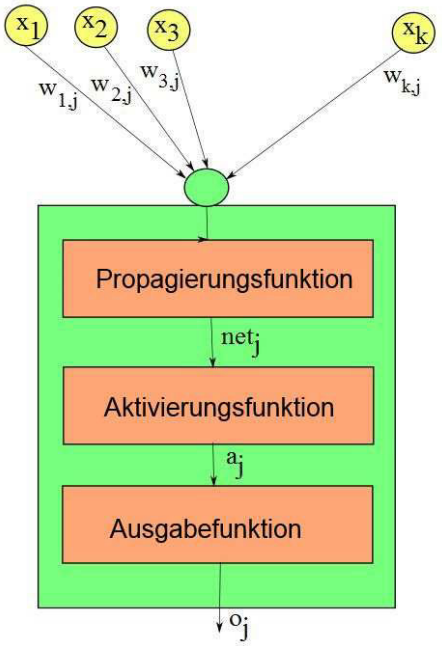


Abbildung 3.4: Neuron, angelehnt an [Ber98]

richtete Verbindungskanten zum Neuron geleitet. Dabei ist der Verbindungskante eine eigene

Funktionalität zugewiesen: Sie verstärkt oder schwächt das eingehende Signal, indem es mit der jeweiligen Wichtung, die mit der Kante verbunden ist, multipliziert wird.

Des Weiteren besitzt jedes Neuron einen Bias-Wert Θ_j. Er stellt einen Schwellenwert dar. Zusammen mit den gewichteten Eingangssignalen geht er in die Berechnung des Eingabewerts net_j ein, der mit Hilfe der Propagierungsfunktion berechnet wird:

$$net_j = \sum x_i w_{i,j} + \Theta_j$$

Der berechnete Eingabewert geht im nächsten Schritt in die Aktivierungsfunktion ein, mit deren Hilfe die aktuelle Aktivierung des Neurons berechnet wird.

Dabei kann die Aktivierung eine reelle Zahl, die auf ein bestimmtes Intervall beschränkt ist, oder eine diskrete Zahl sein. Eine wichtige Aktivierungsfunktionen werden im Folgenden kurz beschrieben.

Die Identitätsfunktion (Die Aktivierung ist identisch mit der Eingabe) ist die Aktivierungsfunktion der Eingabeeinheiten, da diese nur dazu dienen, die Eingangssignale an andere Netzeinheiten zu verteilen. Im Gegensatz zu anderen Netzeinheiten besitzen Eingabeneuronen lediglich einen Eingabewert.

Eine weitere Aktivierungsfunktion ist die binäre Schwellenfunktion:

$$f(net_j) = \begin{cases} 1 & , falls\ net_j \geq \Theta_j \\ 0 & , falls\ net_j < \Theta_j \end{cases}$$

Eine Problematik bei dieser Aktivierungsfunktion ist ihre Unstetigkeit, da unstetige Funktionen nicht differenzierbar sind. Differenzierbarkeit ist dabei für die Lernalgorithmen von Bedeutung.

Eine gängige Aktivierungsfunktion ist die sigmoide Funktion, deren Wert in einen kontinuierlichen Bereich zwischen 0 und 1 fällt. Ein Beispiel für eine sigmoide Funktion ist in Abbildung 3.5 gezeigt.

Die Ausgabe o_j eines Neurons wird mit Hilfe der Ausgabefunktion, die auf die Aktivierung angewandt wird, berechnet. Üblicherweise wird die Identitätsfunktion für die Ausgabefunktion

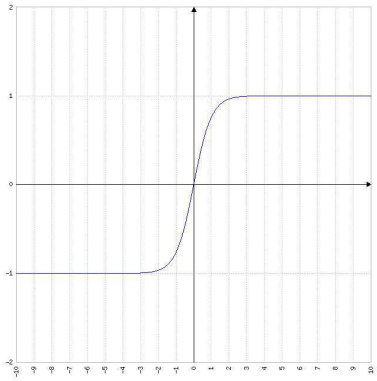


Abbildung 3.5: Beispiel einer sigmoiden Funktion.

verwendet.

Weitere Informationen über den Aufbau eines Neurons sind in [Cal03] und [Alp08] zu finden.

3.2.2 Topologie eines Neuronalen Netzes

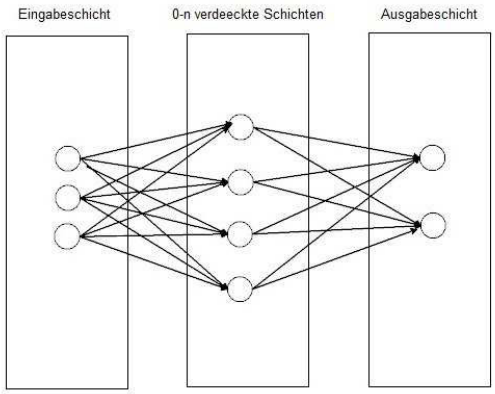


Abbildung 3.6: Beispieltopologie eines Feed-Forward Netzes

Im Neuronalen Netz werden die Neuronen ebenenweise angeordnet. Die Eingaben an das Neuronale Netz werden dabei von der untersten Schicht – der Eingabeschicht – entgegengenommen. Ausgaben des Netzes werden über die Ausgabeschicht nach Außen geliefert. Zwischen diesen beiden Schichten können sich beliebig viele verdeckte Schichten befinden.

Anhand ihres Konnektivitätsmusters, welches beschreibt, auf welche Weise die einzelnen

Neuronen miteinander verbunden sind, können Netze weiter unterschieden werden.

Ein Beispiel für eine Klasse von Neuronalen Netzen sind die vorwärts gerichteten Netze („*feedforward*“), bei denen die Verbindungen innerhalb des Netzes nur in Richtung Ausgabe gehen. Ebenfalls wäre es in solch einem Netz denkbar, dass einzelne Verbindungen dahinter liegende Schichten überspringen. Dies wird als „*shortcut-connections*“ bezeichnet.

Eine andere Klasse von Neuronalen Netzen sind die sogenannten rückgekoppelten Netze. Bei ihnen sind die Verbindungen nicht nur in Richtung Ausgabeschicht gerichtet. Möglich wäre hier, dass ein Neuron seine Ausgabe als Teil der Eingabe der nächsten Berechnung benutzt. Ebenfalls möglich sind die Varianten, dass die Ausgabe an Neuronen derselben oder der vorangehenden Schicht geleitet wird.

3.2.3 Lernen im Neuronalen Netz (vorwärts gerichtete Netze)

Das Lernen in vorwärts gerichteten Netzen geschieht, indem die Verbindungsgewichte zwischen den einzelnen Neuronen an das konkrete Problem angepasst werden. Beim überwachten Lernen hat man eine Menge von Trainingsbeispielen, die aus Eingabevektoren und dazugehörigen Ausgaben $t_{p,j}$ bestehen.

Zu Beginn des Lernverfahrens werden die Verbindungsgewichte mit zufälligen Werten belegt. Danach werden sie schrittweise an die zu trainierenden Daten angepasst. Pro Lernschritt wird dabei zunächst eine Eingabe aus den Trainingsdaten an das Netz gegeben, welches mit Hilfe seiner aktuellen Verbindungsgewichte eine Ausgabe $o_{p,j}$ erzeugt.

Da zu jeder Eingabe in den Trainingsdaten eine gewünschte Ausgabe vorhanden ist, kann in Anschluss der Fehler zwischen der vom Netz berechneten Ausgabe und der gewünschten Ausgabe berechnet werden. Hieraus lässt sich wiederum ein Gesamtfehler E über alle Trainingsdaten berechnen:

$$E = \sum_{p} (t_{p,j} - o_{p,j})^2$$

p steht hierbei für ein Trainingsbeispiel. $t_{p,j}$ ist die dazugehörige gewünschte Ausgabe und $o_{p,j}$ ist die tatsächliche Ausgabe von Neuron j.

Das Ziel des Lernverfahrens ist es, diesen Fehler zu minimieren. Dies stellt ein Optimie-

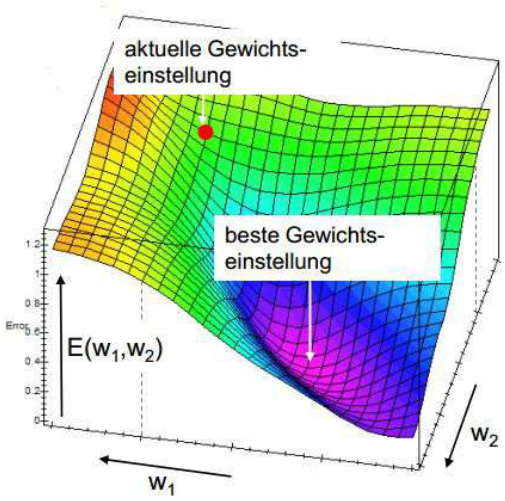


Abbildung 3.7: Abweichungen der Ausgaben von den Trainingsdaten für zwei Gewichte. Quelle: [Mei08]

rungsverfahren dar, welches man z.B. durch Gradientenverfahren lösen kann, wenn man für die Aktivierungsfunktion der Neuronen eine differenzierbare Funktion gewählt hat.

Die Gewichte können nun iterativ durch folgende Formel angepasst werden:

$$\vec{w_{n+1}} = \vec{w_n} - \eta * \vec{\Delta} E$$

Hierbei stellt η einen konstanten Schrittweitenfaktor dar, $\vec{\Delta} E$ ist der Gradient im aktuellen Punkt des Fehlergebirges:

$$\vec{\Delta} E = \frac{\delta E(w_{i,j})}{\delta w_{i,j}}$$

.

Bei dieser Art des Lernen können verschiedene Probleme auftreten, die jedoch durch eine geeignete Modifikation des Lernalgorithmus überwunden oder abgemindert werden können.

Zwei mögliche Probleme werden hier beispielhaft beschrieben:

Lokale Minima: Die Schrittweite wird bei der Suche nach einem guten Minimum durch den Schrittweitenfaktor η und den aktuellen Gradienten bestimmt. Wenn der Algorithmus ein lokales Minimum gefunden hat, ist der Gradient = 0, wodurch die Schrittweite ebenfalls den Wert 0 annimmt. Das lokale Minimum wird nicht mehr verlassen und ein möglicherweise vorhandenen globales Minimum nicht gefunden.

Verlassen guter Minima: Befindet sich der Algorithmus am Rand eines guten Minimums, könnte der Gradient an dieser Stelle so hoch sein, dass das Minimum bei der nächsten Berechnung übersprungen wird (da ein hoher Gradient eine hohe Schrittweite bewirkt) und jenseits dieser möglicherweise guten Lösung weiter gesucht wird.

Für tiefer gehende Informationen zum Lernen in Neuronalen Netzen sei auch hier auf [Cal03] und [Alp08] verwiesen.

3.3 Modulare Neuronale Netze

Herkömmlichen monolithische Neuronalen Netze werden mittlerweile in vielen praktischen Anwendungen angewendet, bei denen kein explizites Wissen vorhanden ist, sondern dem System durch Beispieldaten das gewünschte Verhalten antrainiert wird. Dabei eignen sie sich gut, um Funktionen zu approximieren und Muster in Daten zu entdecken. Diese Eigenschaften können zum Beispiel zur Klassifikation von Datenvektoren genutzt werden.

In Bezug auf die Funktionsapproximation sind die herkömmlichen monolithischen Neuronalen Netze prinzipiell in der Lage, jede Funktion zu approximieren. Praktisch lassen sich aufgrund der globalen Sicht von monolithischen Netzen jedoch besonders gut Funktionen annähern, deren Funktionsverlauf möglichst glatt ist, während solche, bei denen sich der Funktionswert lokal relativ abrupt ändert, schwerer zu modellieren sind. Bei solchen Funktionen muss der Funktionsverlauf dort, wo das Funktionsgebirge stark zerklüftet ist, durch sehr viele Trainingsdaten beschrieben werden, um eine ausreichende Generalisierung sicherzustellen [JJ93].

Neben den monolithischen Neuronalen Netzen hat sich mit den modularen Neuronalen Netzen eine Möglichkeit entwickelt, um z.B. Regressions- oder Klassifikationsprobleme zu lösen. Hierbei treten sie besonders bei der Modellierung von Funktionen mit scharfen Änderungen im Funktionsbereich in Konkurrenz zu den herkömmlichen monolithischen Netzen[FH96].

Auch wenn es keine einheitliche Definition davon gibt, welche Eigenschaften ein modulares Neuronales Netz ausmachen, so fasst Azam in seiner Arbeit [Aza00] die gängigsten Meinungen in folgender Definition zusammen:

Definition 1. *„A neural Network is said to be modular if the computation performed by the network can be decomposed into two or more modules (subsystems) that operate on distinct inputs without communicating with each other. The outputs of the modules are mediated by an integrating unit that is not permitted to feed information back to the modules. In particular, the integreating unit decided both (1) how the modules are combined to form the final output, and (2) which modules should learn which training patterns.“*

Allerdings beschreibt er auch, dass es momentan nur sehr wenige Modelle gibt, die alle in dieser Definition geforderten Eigenschaften erfüllen, jedoch eine Vielzahl von Architekturen für modulare Neuronale Netze im weiteren Sinne.

Bei dem Versuch, für bestimmte Aufgaben zuverlässigere Architekturen als die klassische

monolithische zu entwickeln, haben sich nach [Aza00] zwei verschiedenen Herangehensweisen entwickelt, wie Probleme durch eine Menge von Neuronalen Netzen gelöst werden können:

1. **Der Ensemble-basierte Ansatz:** Es werden mehrere Neuronale Netze für das Lösen derselben Problemstellung trainiert und deren Einzellösungen zu einer Gesamtlösung kombiniert.

2. **Der modulare Ansatz:** Das Problem wird in mehrere Teilprobleme zerlegt, für welche Spezialisten trainiert werden.

Beide Ansätze werden im Folgenden kurz vorgestellt.

3.3.1 Der Ensemble-basierte Ansatz

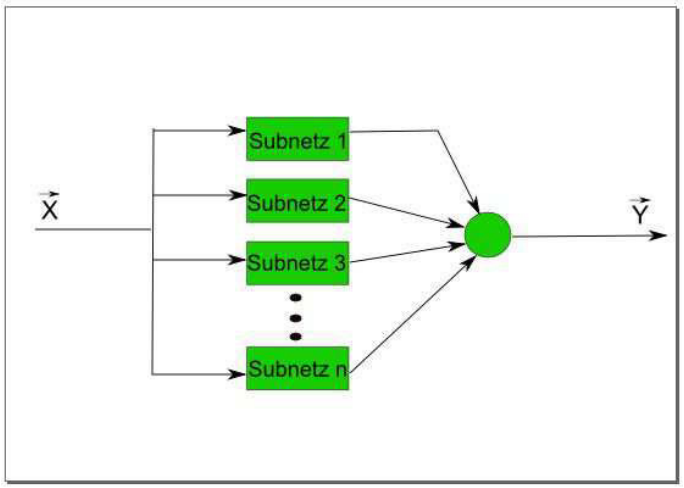


Abbildung 3.8: Möglicher Aufbau eines modularen Neuronalen Netzes nach dem Ensemble-basierten Ansatz

Die Hauptidee des Ensemble-basierten Ansatzes besteht darin, dass eine Menge von verschiedenen, bereits trainierten Neuronalen Netzen zur Lösung eines Problems genutzt wird. Dabei sind die einzelnen Mitglieder des Ensembles so trainiert, dass sie alle dieselbe Aufgabe lösen sollen. Dabei beeinflussen sich die einzelnen Module nicht gegenseitig in ihren Berechnungen.

Beruhend auf der Annahme, dass eine optimale Kombination der Ausgaben aller Mitglieder eine bessere Ausgabe des Gesamtsystems erzeugt, als wenn man sich alleine auf die Antwort

eines einzelnen Netzes verlässt, werden anschließend die einzelnen Ausgaben kombiniert. Auf diese Weise sollen zuverlässigere und genauere Ausgaben erreicht werden, als wenn man sich lediglich für die Ausgabe eines Netzes verlässt (bei dem man im Vorfeld herausgefunden hat, dass es das durch die Beispiele definierte Modell am besten annähert).

Wesentliche Aufgaben bei dieser Herangehensweise sind das Erzeugen einer Menge von Neuronalen Netzen, die sich in ihren Eigenschaften unterscheiden können, und das Festlegen einer Methode, wie die einzelnen Ausgaben zum Schluss kombiniert werden können.

Mögliche Merkmale, in denen sich einzelne Mitglieder unterscheiden könnten, sind beispielsweise die Netztopologie und der Lernalgorithmus. Es ist aber auch denkbar, dass verschiedene Mitglieder mit verschiedenen Trainingsdatensätzen trainiert werden.

Ebenso sind bei der Generierung der Gesamtlösung aus den Einzelausgaben mehrere Methoden möglich und dokumentiert. Einige von ihnen sind Mittelwertbildung, gewichtete Mittelwertbildung und verschiedene Voting-Mechanismen. Weiterhin kann man hier die Evidenztheorie von Dempster und Shafer nennen, die ebenfalls dazu dient, aus einzelnen Ergebnissen verschiedener Quellen unter Berücksichtigung der Glaubwürdigkeit dieser Quellen ein gesamtes Ergebnis zu konstruieren.

Ein Möglicher Aufbau eines modularen Neuronalen Netzes nach dem Ensemble-basierten Ansatz ist in Abbildung 3.8 zu sehen.

3.3.2 Der modulare Ansatz

Der modulare Ansatz ist eine Erweiterung des Grundsatzes *Devide and Conquer*. Bei diesem geht es darum, ein großes, komplexes Problem solange in Teilprobleme aufzuspalten, bis diese lösbar werden. In Anschluss können die Lösungen der Teilprobleme wieder zusammengesetzt werden, um eine Gesamtlösung für das ursprüngliche Problem zu generieren.

Bei diesem Ansatz sind die Module folglich Spezialisten, die dazu trainiert wurden, eine spezielle Aufgabe zu lösen. Hierzu können sie eine bestimmte geeignete Topologie aufweisen oder durch andere Maßnahmen (unterschiedliche Lernmethoden etc.) den Besonderheiten des jeweiligen Teilproblems angepasst werden. Auch hier beeinflussen sich die einzelnen Module typischerweise nicht gegenseitig.

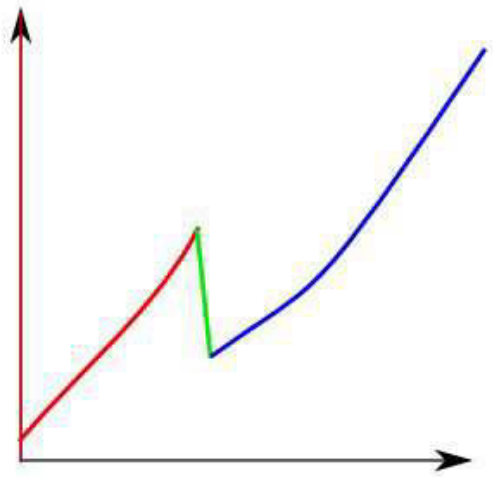


Abbildung 3.9: Approximation einer Funktion durch mehrere Spezialisten

In Abbildung 3.9 wird gezeigt, wie eine Funktion durch mehrere Spezialisten approximiert werden kann. Die Funkion ist in verschiedene Abschnitte (in der Abbildung farblich dargestellt) unterteilt, die jeweils durch einen anderen Spezialisten angenähert werden. Auf diese Weise kann eine Funktion aus mehreren Teilen zusammengesetzt werden.

Genau wie bei dem Ensemble-basierten Ansatz gibt es auch bei dieser Herangehensweise zwei Hauptaufgaben, die beim Entwurf eines solchen modularen Neuronalen Netzes zu beachten sind. Die erste besteht darin, zu entscheiden, wie das Problem zerlegt und auf eine angemessene Anzahl von Spezialisten aufgeteilt werden soll. Die zweite ist das Zusammenfügen der einzelnen Ausgaben der Spezialisten.

Ein Beispiel für ein Problem, welches auf diese Weise gelöst werden könnte, wäre ein Klassifikator, der handgeschriebene Ziffern erkennen soll.

Hierzu könnten ähnliche Ziffern zunächst zu Clustern zusammengefasst werden. Auf diese Weise könnten zunächst die Cluster {0,6,8},{1,2,3,7} und {4,5,9} von ähnlichen Ziffern gebildet werden. Für jedes Cluster wird nun ein Expertennetz trainiert, welches nur die Trainingsdaten der Ziffern aus seinem Cluster erlernt.

Neben den Expertennetzen wird weiterhin ein Weichennetz benötigt, welches die Trainingsdaten von allen Ziffern zum Training benutzt. Es lernt, seine Eingaben einem der Cluster und

somit einem der Expertennetze zuzuordnen.

Wenn das modulare Netz nach dem Training die Ziffer 3 erkennen soll, wird die Eingabe zunächst an das Weichennetz geleitet. Dieses berechnet, zu welchem Cluster die Eingabe gehört (Cluster {1,2,3,7}). Im nächsten Schritt wird die Eingabe nun an das Expertennetz geleitet, das für dieses Cluster zuständig ist. Es berechnet nun die endgültige Ausgabe, nämlich dass die Ziffer in der Eingabe eine 3 ist.

Wagner [Wag99] beschreibt in seiner Arbeit zwei grundsätzliche Möglichkeiten, die Trainings-

	Eingabe			Ausgabe	
	Dimension 1	Dimension 2	Dimension 3	Ausgabe 1	Ausgabe 2
Beispiel 1	0	0	0	0	1
Beispiel 2	0	0	1	1	0
Beispiel 3	0	1	0	0	1
Beispiel 4	0	1	1	1	0
Beispiel 5	1	0	0	0	1
Beispiel 6	1	0	1	1	0
Beispiel 7	1	1	0	1	0

	Eingabe			Ausgabe	
	Dimension 1	Dimension 2	Dimension 3	Ausgabe 1	Ausgabe 2
Beispiel 1	0	0	0	0	1
Beispiel 2	0	0	1	1	0
Beispiel 3	0	1	0	0	1
Beispiel 4	0	1	1	1	0
Beispiel 5	1	0	0	0	1
Beispiel 6	1	0	1	1	0
Beispiel 7	1	1	0	1	0

Abbildung 3.10: Zerlegung der Daten in Regionen (oben) und Zerlegung der Daten entlang der Dimensionen (unten). Angelehnt an [Wag99]

daten auf Spezialisten aufzuteilen:

Aufteilung nach Regionen: Bei der Aufteilung nach Regionen werden die Trainingsbeispiele in Gruppen eingeteilt. Jede Gruppe wird dann von einem eigenen Modul trainiert. Diese Aufteilung ist insbesondere dann sinnvoll, wenn sich die anzunähernde Funktion in unterschiedlichen Regionen stark unterscheidet.

Zerlegung entlang der Dimensionen: Bei der Zerlegung der Daten entlang der Dimensionen sieht jedes Expertennetz alle Trainingsbeispiele, von diesem jedoch nur bestimmte Eigenschaften.

Mögliche Vorteile von modularen Neuronalen Netzen, deren Expertennetze mit regional getrennten Trainingsdaten trainiert wurden sind im Folgenden aufgelistet. [JJ93] [Aza00] [Wag99]

Reduzierung der Komplexität: Bei komplexeren Problemen werden bei monolithischen Neuronalen Netzen sehr viele Neuronen benötigt, um diese ausreichend anzunähern. Mit dem Hinzufügen von weiteren Neuronen oder weiteren verdeckten Schichten wächst jedoch auch die Anzahl der Verbindungsgewichte drastisch. Modulare Neuronale Netze können diese Probleme umgehen, da die Expertennetze kleine, einfache Teilaufgaben trainieren. Da die spezialisierten Module weniger Verbindungsgewichte trainieren müssen, reduziert sich bei modularen Neuronalen Netzen ebenfalls die Zeit, die zum Trainieren der Gewichte benötigt wird. Auch bei der Berechnung müssen später weniger Verbindungsgewichte mit den Eingaben multipliziert werden, was zu einer schnelleren Berechnung der Ausgaben führt.

Geringere Anfälligkeit für zeitliche Interferenz: Zeitliche Interferenz („*temporal crosstalk*“) ist ein Effekt, der auftreten kann, wenn ein monolithisches Neuronales Netz während des Trainings zunächst mit Trainingsbeispielen aus einer Region und im Anschluss mit Trainingsbeispielen überwiegend aus einer anderen Region trainiert wird. Unterscheiden sich diese beiden Regionen stark voneinander, so wird das Netz seine während des Trainings des ersten Datenblocks erlernten Gewichtseinstellungen beim Trainieren des zweiten Datenblocks ändern. Auf die Weise passt es sich an die neuen Anforderungen dieser Daten, die sich stark von den zuerst erlernten Daten unterscheiden, an. Hierdurch werden jedoch die alten Einstellungen verändert und das Netz „vergisst“ die Trainingsdaten aus der zuerst gelernten Region.

Dies führt zu einer längeren Dauer des Trainings und gegebenenfalls zu einer schlechteren Generalisierungsfähigkeit. Bei entsprechend angepassten modularen Netzen hingegen werden üblicherweise – sofern die Trainingsdaten nach Regionen getrennt werden – unterschiedliche Module für das Lernen zweier unterschiedlicher Regionen verwendet. Auf diese Weise sind sie immun gegenüber zeitlicher Interferenz.

Bessere Generalisierungsfähigkeit: Ein Neuronales Netz verfügt im Allgemeinen dann über eine gute Fähigkeit zu generalisieren, wenn es gerade so flexibel ist, dass es die anzunähernde Funktion gerade noch darstellen kann.

Ist die zu approximierende Funktion aus einfachen Teilen aufgebaut, jedoch insgesamt äußerst komplex, generalisiert ein aus einfachen Modulen aufgebautes Netz, die für eben diese einfachen Teile der Funktion zuständig sind, in der Regel besser als ein monolithisches Netz. Ein weiterer Vorteil ist, dass durch eine bessere Generalisierungsfähigkeit auch insgesamt weniger Trainingsdaten benötigt werden.

Nutzen von Vorwissen: Es ist möglich, bei der Zerlegung in Module a priori Wissen in der Architektur des Systems einfließen zu lassen, welche eine große Auswirkung auf die Funktion des Netzes hat.

3.3.3 Halbautomatische Aufteilung bei regional-modularen Netzen

Wagner [Wag99] unterteilt in seiner Arbeit regional-modulare Netze noch einmal in 3 Kategorien:

Manuelle Aufteilung: Netze, bei denen die Trainingsdaten händisch in kleinere Gruppen zerlegt wurden.

Halbautomatische Aufteilung: Netze, bei denen die Trainingsdaten automatisch in Gruppen eingeteilt wurden, die jeweiligen Expertennetze jedoch unabhängig voneinander trainiert werden.

Automatische Aufteilung: Verfahren, bei denen Gewichte der Expertennetze und Weichennetze gleichzeitig trainiert werden. Hier sind insbesondere Expertenmischsysteme wie HME („*Hierarchical Mixture of Experts Model*“) und EHME („*Extended Hierarchical Mixture of Experts Model*“) zu nennen [Aza00].

Ein Beispiel für einen halbautomatischen Ansatz ist das von Bennani beschriebene System, welches entscheiden soll, welchem von 102 Sprechern eine bestimmte Sprachprobe zuzuordnen ist. Aufgrund der großen Anzahl von zu unterscheidenden Personen ist dies für ein monolithisches Netz eine schwer zu lösende Aufgabe [Ben95].

Daher verwendet er ein modulares Neuronales Netz, bei dem die Sprecher durch ein k-means Clusteringverfahren (siehe Abschnitt 3.5.1) anhand von stimmlichen Merkmalen in 16 Teilgruppen eingeordnet werden. Im Anschluss an diesen Schritt wurde jedes Expertennetz mit den Daten einer der 16 Gruppen trainiert, während ein weiteres Modul – das Weichennetz – für die Generierung der Gesamtausgabe des Systems verantwortlich ist. Hier schlägt Bennani zwei Methoden vor, wie die Teilergebnisse der einzelnen Experten zu einem Gesamtergebnis kombiniert werden können. Diese sind in Abbildung 3.11 zu sehen.

Bei der ersten Möglichkeit wird die Eingabe $\vec{x}$ an alle Expertenmodule und das Weichennetz geleitet. Die Ausgaben der Expertenmodule werden dann durch das Weichennetz gewichtet und zu einer Gesamtausgabe verrechnet.

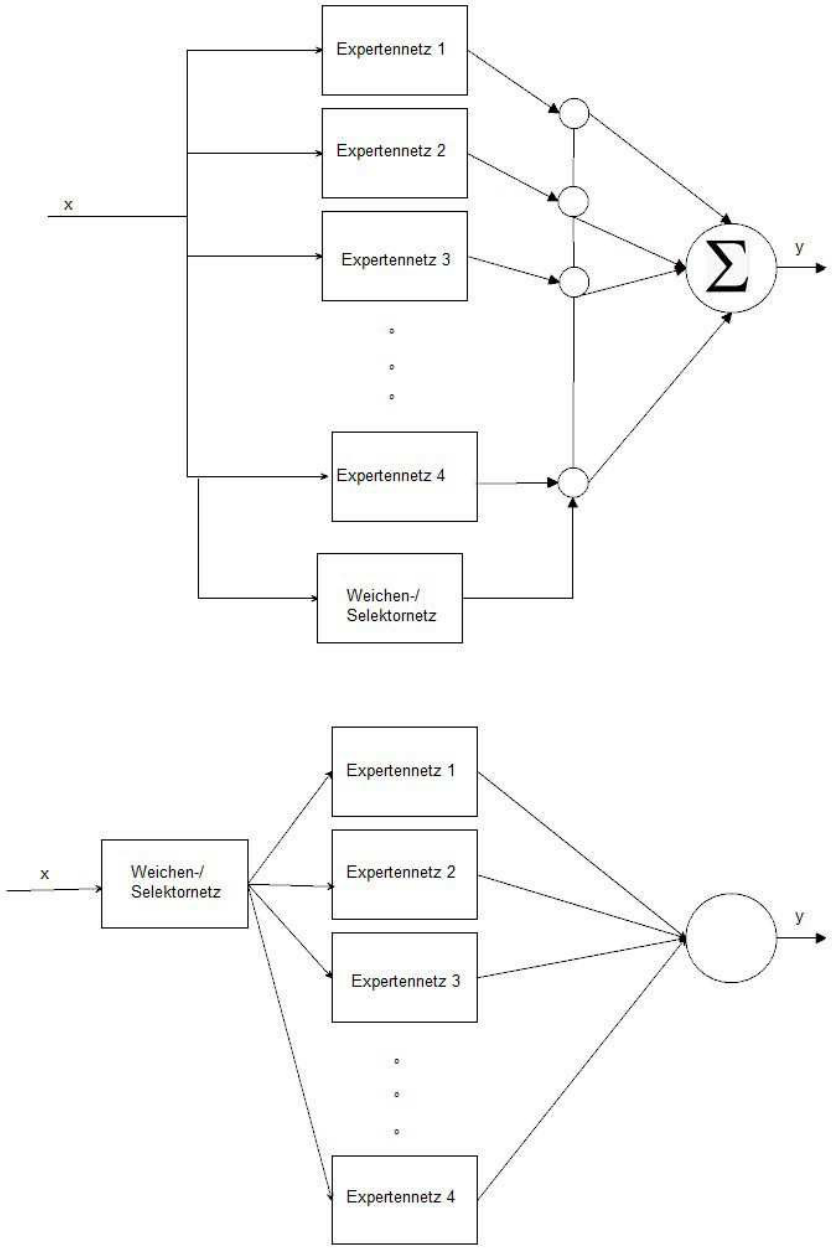


Abbildung 3.11: Mögliche Experten-Netz Architekturen. Oben: Berechnung einer Ausgabe durch alle Expertennetze und Wichtung der Ausgaben durch ein Weichennetz. Unten: Auswahl eines Expertennetzes, welches für die Ausgabe zuständig ist, durch das Weichennetz. Angelehnt an [Wag99]

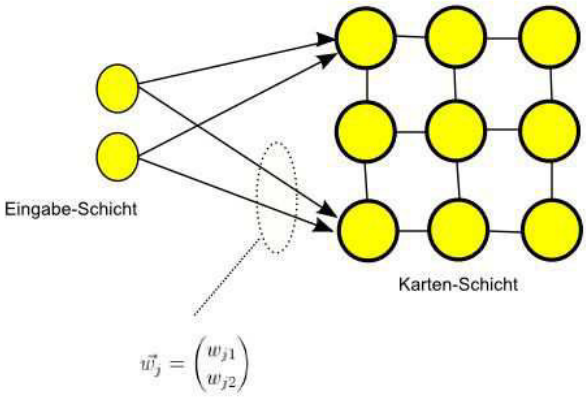


Abbildung 3.12: Skizzenhafte Architektur einer SOM, angelehnt an [LC12]

Bei der zweiten Möglichkeit wird die Eingabe zunächst an das Weichennetz geleitet, welches daraufhin entscheidet, welcher Gruppe diese zuzuordnen ist. Danach wird die Eingabe an das Expertennetz weitergeleitet, welches für eben diese Gruppe zuständig ist.

Der Vorteil der zweiten Möglichkeit gegenüber der ersten ist, dass sie signifikant weniger Rechenzeit benötigt, da nur ein Experte anstatt aller 16 eine Ausgabe berechnet. Allerdings besteht hier nicht die Möglichkeit, dass ein möglicher Fehler des Weichennetzes durch Einbeziehung aller Experten ausgeglichen wird.

3.4 Selbstorganisierende Karte

Selbstorganisierende Karten (Self Organizing Map – SOM) sind eine weitere Form von Neuronalen Netzen. Zwei wesentliche Merkmale von SOMs sind, dass sie erstens unüberwachtes Lernen verwenden, wodurch sie sich besonders gut zur Clusteranalyse eignen, und zweitens, dass sie die räumliche Anordnung der Neuronen zueinander ausnutzen, da Neuronen das Verhalten ihrer benachbarten Neuronen beeinflussen.

Self Organizing Maps bestehen aus zwei Schichten: der Eingabe-Schicht und der Karten-Schicht. Dabei sind die Neuronen der Eingabe-Schicht vollständig mit den Neuronen der Karten-Schicht vernetzt . Die Neuronen der Karten-Schicht sind in einem zweidimensionalen, rechteckigen Gitter angeordnet. Eine Skizze des Aufbaus einer SOM ist in Abbildung 3.12 zu sehen.

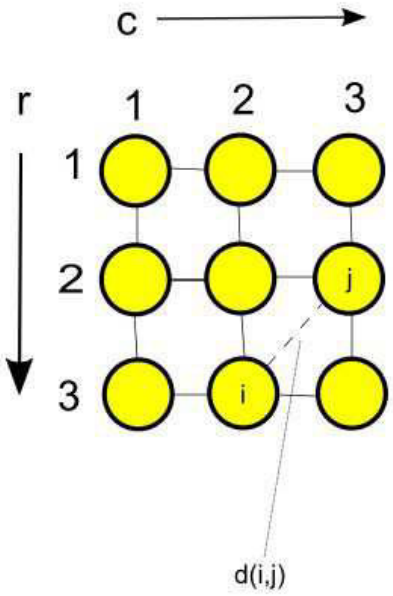


Abbildung 3.13: Der topologische Abstand zweier Neuronen, angelehnt an [Mei08]

Die Aufgabe der Eingabe-Schicht ist es, die Eingaben an die Neuronen der Karten-Schicht weiterzuleiten. Hierdurch geschieht eine Abbildung von einem höher dimensionalen Raum in einen niedrig dimensionalen Raum.

Die Lage eines Neurons der Karten-Schicht lässt sich auf zwei Arten beschreiben: durch seine Lage im zweidimensionalen Neuronengitter und durch seine Lage im Eingaberaum. Letztere wird durch die Verbindungsgewichte der Verbindungen beschrieben, die von der Eingabe-Schicht zum Neuron der Karten-Schicht führen.

Beim Training der SOM werden die Trainingsdaten dem Netz immer wieder in ungeordneter Reihenfolge präsentiert. Pro Trainingsmuster wird dann folgender Ablauf wiederholt: Nachdem das Gewinnerneuron gefunden wurde, werden im nächsten Schritt die Gewichte zwischen der Eingabe-Schicht und der Karten-Schicht verändert. Dabei wird das Gewinnerneuron am stärksten vom aktuell präsentierten Muster beeinflusst. Die aktuelle Eingabe nimmt jedoch auch Einfluss auf die anderen Neuronen der Karten-Schicht. Neuronen, die näher am Gewinnerneuron liegen, werden dabei stärker beeinflusst als Neuronen, die weiter von diesem entfernt sind. Hierzu wird der topologische Abstand berechnet, welches der Abstand zweier Neuronen im zweidimensionalen Gitter ist (siehe Abbildung 3.13). Die topologische Distanz

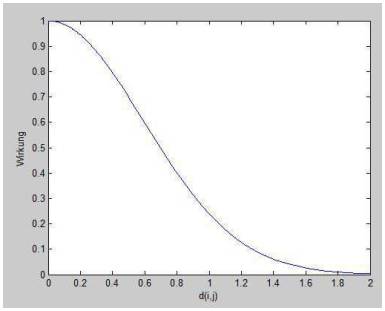


Abbildung 3.14: Wirkungsfunktion

$d_{i,j}$ zweier Neuronen i,j wird nun genutzt, um die Wirkung von i auf j zu berechnen. Hierzu wird die topologische Distanzfunktion genutzt, welche Beispielhaft in Abbildung 3.14 gezeigt wird:

$$\tau_{ij} = e^{-\frac{d_{ij}^2}{2\sigma^2}}$$

Die Veränderung der Verbindungsgewichte jedes Neurons wird dann durch folgende Formel berechnet:

$$w_j(t+1) = w_j + \eta * \tau_{ij} * (\vec{x} - \vec{w_j}(t))$$

η ist hierbei ein Schrittweitenfaktor, der beeinflusst, wie stark ein einzelnes Trainingsmuster die Neuronen beeinflusst.

Durch die Veränderung der Gewichte werden die Verbindungsgewichte eines Neurons in Richtung der Eingabe $\vec{x}$ verändert. Die Stärke dieser Veränderung ist von der topologischen Distanz $d_{i,j}$ zwischen Neuron i und Gewinnerneuron j abhängig.

Die Adaption der Gewichte bewirkt, dass sich nach und nach Erregungszentren für Musterklassen herausbilden.

Eine trainierte SOM kann als eine Art „Datenlandkarte“ angesehen werden. Es wird eine räumliche Anordnung der Datenpunkte präsentiert, wobei ähnliche Datenpunkten von benachbarten Bereichen der Karte repräsentiert werden. Aus diesem Grund eigenen SOMs sich zur Visualisierung von Daten und zur Datenkompression [Mer08]. Möchte man Daten mit Hilfe von SOMs clustern, so müssen die Neuronen einer trainierten SOM anhand ihrer Verbindungsgewichte zu Clustern zusammengefasst werden. Auf diese Weise betreibt man ein

zweistufiges Clustering [VA00].

Eine gute Übersicht zu dem Aufbau, dem Trainings und der Arbeitsweise vom Selbstorganisierenden Karten findet man in [LC12].

3.5 Clusteranalyse

Ziel der Clusteranalyse ist es, in Daten Gruppen (Cluster) von ähnlichen Daten zu bilden. Dabei sollen Cluster so gebildet werden, dass folgende Kriterien möglichst gut erfüllt werden:

1. Objekte, die demselben Cluster zugeordnet wurden, sollen sich möglichst ähnlich in ihren Eigenschaften sein.
2. Objekte, die verschiedenen Clustern zugeordnet wurden, sollen sich möglichst stark in ihren Eigenschaften unterscheiden.

Eine große Bandbreite an Möglichkeiten, Ähnlichkeiten zwischen Clustern und innerhalb eines Clusters zu messen, lässt hier jedoch auch viel Raum für verschiedene Interpretationen dieser Kriterien [VA00].

Bei der Bildung von Clustern kann man zwei Arten unterscheiden: Cluster können hierarchisch oder durch partitionierende Verfahren gebildet werden. Bei der hierarchischen Clusterung setzen sich Cluster aus Subclustern zusammen, sodass eine Baumstruktur entsteht. Diese muss im Anschluss an einer geeigneten Stelle geteilt werden, um die Objekte zu Clustern zusammenzufassen. Die Anzahl der Cluster ist hier nicht festgelegt.

Bei den partitionierenden Verfahren steht die Anzahl der Cluster von Anfang an fest und es werden lediglich Objekte zwischen den einzelnen Clustern verschoben.

Es gibt verschiedene Algorithmen zur Bestimmung der Cluster, von denen zwei in dieser Arbeit verwendete im Folgenden vorgestellt werden.

3.5.1 K-Means

K-Means gehört zu den partitionierenden Verfahren. Bei diesem Algorithmus werden die Daten in k disjunkte Cluster eingeteilt. K ist dabei ein vordefinierter Wert. Jedes einem Cluster zuzuordnende Objekt wird hier als Punkt in einem n-dimensionalen Raum betrachtet, wobei n

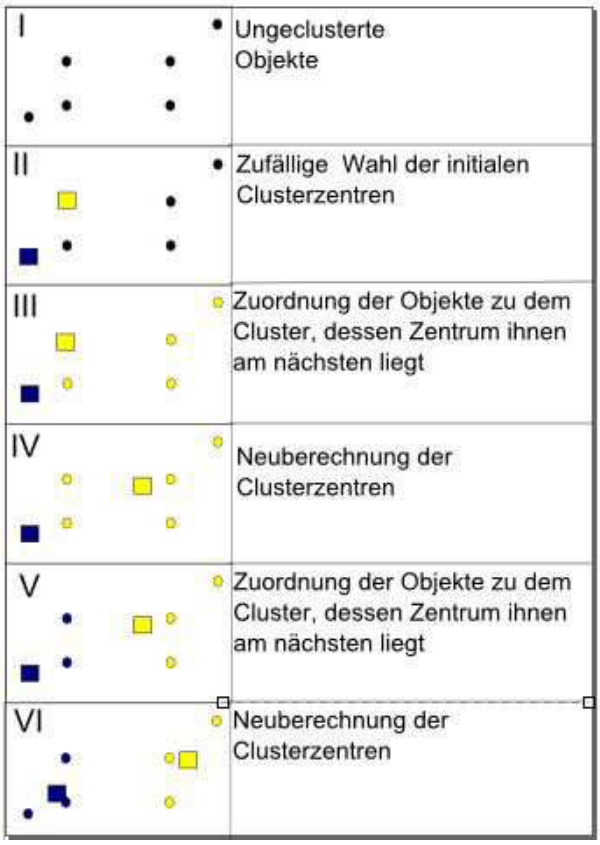


Abbildung 3.15: K-Means Algorithmus

die Anzahl der Objekteigenschaften ist, nach denen die Cluster gebildet werden. Jedes Cluster wird durch seinen Schwerpunkt (Centriod) und durch die ihm zugewiesenen Objekte definiert.

Der Algorithmus lässt sich folgendermaßen beschreiben:

1. Zu Beginn des Algorithmus werden k zufällige Punkte als Schwerpunkte der initialen Cluster festgelegt.
2. Jeder Objektpunkt wird dem Cluster zugeordnet, dessen Schwerpunktpunkt dem Objektpunkt am nächsten liegt.
3. Neuberechnung der Schwerpunkte.
4. Schritt 2 und 3 werden solange wiederholt, bis sich die Schwerpunkte nicht mehr verändern oder eine vorgegebene Anzahl von Iterationen erreicht ist.

Ein möglicher Ablauf des K-Means Algorithmus ist in Abbildung 3.15 dargestellt.

Bei diesem Algorithmus hängt das Ergebnis von der Wahl der initialen Clusterschwerpunkte ab, sodass bei mehreren Durchläufen unterschiedliche Ergebnisse erzielt werden. Aus diesem Grund wird der Algorithmus häufig mehrmals ausgeführt und zum Schluss das beste Ergebnis genommen.

Ein weiterer Nachteil von K-Means ist wie bei allen partitionierenden Verfahren , dass konkrete Annahmen über die Form der Cluster gemacht werden (K-means nimmt zum Beispiel kugelförmige Cluster an) [VA00].

3.5.2 Hierarchische Clusterverfahren

Bei den hierarchischen Verfahren wird eine hierarchische Struktur aufgebaut, bei der sich Cluster aus Subclustern zusammensetzen. Hierzu gibt es zwei Typen von Verfahren: Bei den agglomerativen (bottom up) stellt jedes Objekt zu Beginn ein eigenes Cluster dar. Nun werden immer die sich ähnlichsten dieser Cluster iterativ zu einem größeren zusammengefasst, bis zum Schluss nur noch ein einziges Cluster übrig bleibt.

Bei den divisiven Verfahren (top-down) wird mit genau einem Cluster gestartet, welches alle Objekte enthält. Dieses wird nun iterativ in Subcluster aufgespalten.

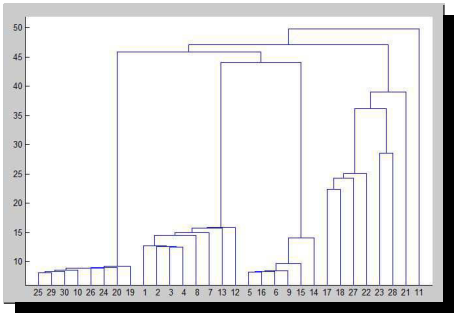
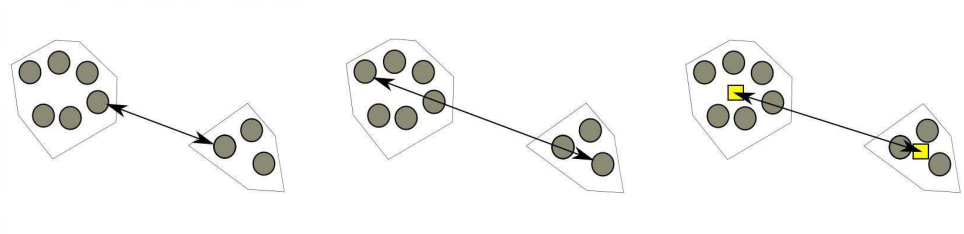


Abbildung 3.16: Dendrogramm

Als Ergebnis erhält man bei diesen Verfahren eine Baumstruktur, die man an einer geeigneten Stelle teilen muss, um eine adäquate Clusterung zu erhalten. Je nachdem, wo man dies macht, sind so verschiedene Clusterungen möglich. Meistens wird dort geteilt, wo sich zwei zusammengefügte Cluster relativ unähnlich sind. Ein Hilfsmittel zum Finden dieser Stellen stellt hier das Dendrogramm dar (siehe Abbildung 3.16). Es zeigt, welche Cluster zusammengefasst werden. Weiterhin gibt die Länge der Kanten von einem Knoten zu seinen Kindern Auskunft darüber, wie ähnlich sich die Cluster sind, die zu diesem einen Knoten zusammengefasst werden.

Eine zentrale Rolle bei diesen Verfahren ist die Wahl eines geeigneten Maßes für die Ähn-



(a) Minimaler Abstand (b) Maximaler Abstand (c) Abstand der Schwerpunkte

Abbildung 3.17: Möglichkeiten Ähnlichkeit zu definieren, angelehnt an [Mer08]

lichkeit zweier Cluster. Einige von diesen sind der minimale Abstand (single linkage), der maximale Abstand (complete linkage) und der Abstand der Centroide der Cluster (centriod distance). Diese sind in Abbildung 3.17 skizzenhaft dargestellt.

Weitere Informationen zu Clustering, sowie zu den hier vorgestellen Clustering-Methoden sind in [Mer08] und [Mat10b] zu finden.

4 Umsetzung

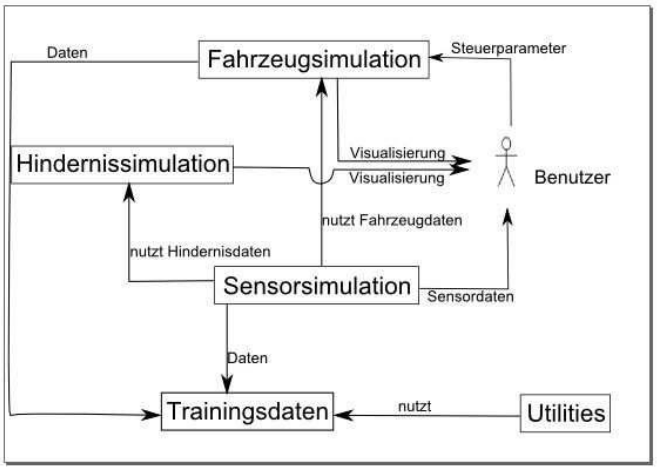
4.1 Die Simulationsumgebung

Ziel dieser Arbeit ist es, Methoden zu untersuchen, wie ein Fahrzeug mit Hilfe von überwachten Lernen in eine Parklücke einparken kann. Hierzu wurde eine Simulationsumgebung verwendet. Die dazu verwendete Simulationsumgebung basiert auf der Arbeit von Rieken[Rie12]. In dieser Arbeit wurde eine Umgebung entwickelt, in der Hindernisse und ein Fahrzeug definiert werden können. Weiterhin ist es möglich, das Fahrzeug durch seine Umwelt zu navigieren – es wird also ausgehend von einem Lenkwinkel und einer Geschwindigkeit des Fahrzeugs die Position des Fahrzeugs zu einem Zeitpunkt berechnet und dargestellt. Außerdem bietet diese Umgebung die Möglichkeit, Sensoren zu definieren, mit deren Hilfe Abstände vom Fahrzeug zu den Hindernissen berechnet werden können.

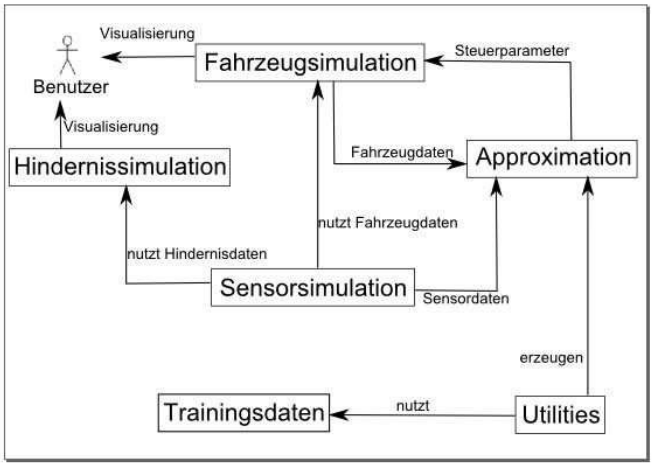
Allerdings besteht das von Rieken entwickelte Programm im Wesentlichen aus einer einzigen Funktion, die für sämtliche Berechnungen zuständig ist. Sensoren und Hindernisse sind fest einprogrammiert und nur mit größerem Aufwand änderbar. Aus diesem Grund wurde das System im Rahmen dieser Arbeit so aufgeteilt, dass bestimmte Aspekte der Funktionalität nun von einzelnen oder einer Menge von Funktionen erfüllt werden. Auf diese Weise lassen sich zum Beispiel weitere Hindernisse oder Sensoren mit weniger Aufwand zur Simulation hinzufügen. Die wichtigsten umgesetzten Bereiche und deren statischen Zusammenhänge sind in Abbildung 4.1 zu sehen. Abbildung 4.1(a) verdeutlicht dabei die statischen Zusammenhänge während der Trainingsdatengenerierung, Abbildung 4.1(b) die statischen Zusammenhänge, wenn das trainierte System einparken soll. Eine Abbildung der GUI wird in Abbildung 4.2 gezeigt.

Die Aufgaben der einzelnen Funktionsbereiche werden im Folgenden kurz erklärt:

Fahrzeugsimulation: Die Fahrzeugsimulation ist für die Simulation des Fahrzeugs verantwortlich, das in die Parklücke navigiert werden soll. Es berechnet ausgehend von Geschwindigkeit und Lenkwinkel die Bewegung des Fahrzeugs.



(a) Generierung der Trainingsdaten



(b) Einparken durch Funktionsapproximation

Abbildung 4.1: Zusammenhänge der umgesetzten Funktionalität

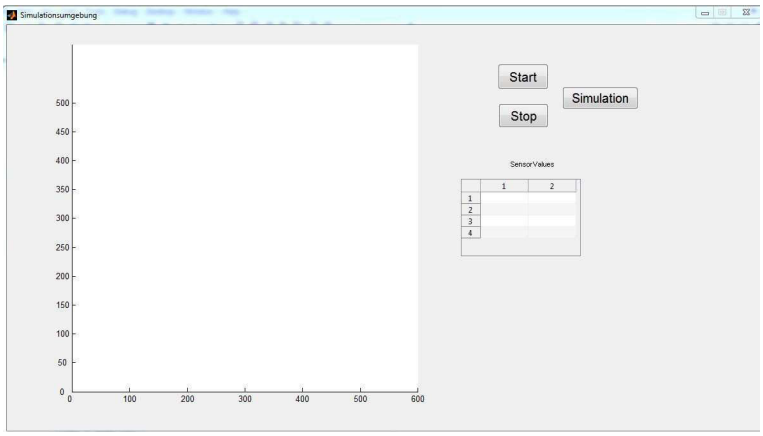


Abbildung 4.2: GUI der Simulationsumgebung

Hindernissimulation: Die Hindernissimulation ist für die Modellierung der Hindernisse zuständig. Durch Hindernisse können parkende Autos, die eine freie Parklücke bilden, oder andere Hindernisse simuliert werden.

Approximation: Dieses Modul ist für die Berechnung der Steuerbefehle (z.B. Geschwindigkeit und Lenkwinkel)zuständig. Es Approximiert die Funktion, die der Agent während des Trainings lernt.

Sensorsimulation: Die Sensorsimulation ist für die Wahrnehmung der Abstände zu den in der Umgebung befindlichen Hindernissen zuständig.

Utilities: Eine Menge von Funktionen, die die Funktionalität zur Vorverarbeitung (z.B. Clusterung und Normierung) der Trainingsdaten zur Verfügung stellen.

Die Simulationsumgebung von Rieken wurde in Matlab geschrieben. Diese Programmiersprache wurde auch für die Weiterentwicklung verwendet, da mit Matlab umfangreiche Toolboxen für Probleme aus den benötigten Bereichen (z.B. Neuronale Netze, Clustering) zur Verfügung stehen.

4.2 Beschreibung des Zustands

In diesem Abschnitt werden verschiedene Möglichkeiten diskutiert, auf welche Weise dem Approximations-Modul, welches die Abbildung vom aktuellen Zustand des Agenten auf eine geeignete Aktion leistet, der aktuelle Zustand des Agenten mitgeteilt werden kann.
Dabei sollten folgende Kriterien nach Möglichkeit erfüllt werden:

Abbildung 4.3: Repräsentation des Zustands durch gemessene Entfernungen

Einfachheit: Um die Leistung des Approximations-Moduls zu steigern, sollten nur Informationen in den Zustand einfließen, die relevant zur Lösung des Problems sind.

Eindeutigkeit: Der Zustand sollte eindeutig und von anderen Zuständen zu unterscheiden sein.

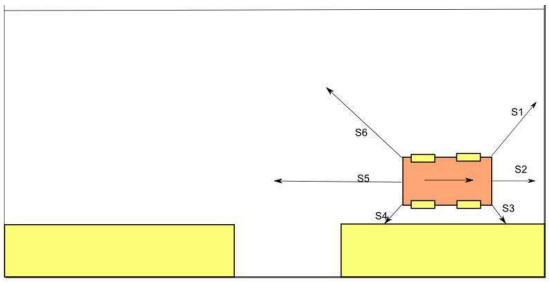
Ähnlichkeit: Benachbarte Zustände sollten zu ähnlichen Ausgaben/Aktionen führen. Diese Forderung ist bei dem hier zu lösenden Einparkproblem nicht immer gewährleistet. Bei einem Einparkvorgang wird in den meisten Fällen auf engem Raum sehr stark in unterschiedliche Richtungen eingeschlagen, um das Fahrzeug in die Parklücke zu manövrieren. Wenn aus dem aktuellen Zustand auf den Lenkwinkel geschlossen werden soll, kann es sein, dass dieser sich durch die Natur des Einparkvorgang abrupt von einem Zustand zum nächsten ändert (z.B. Lenken in einem stehenden Fahrzeug).

Störungsempfindlichkeit: Die gewählte Zustandsrepräsentation sollte unempfindlich auf Störungen reagieren.

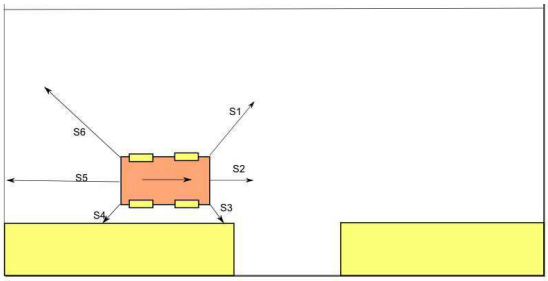
In dieser Arbeit wurden dabei zwei unterschiedliche Möglichkeiten der Zustandsdarstellung untersucht. Eine Möglichkeit ist es, den aktuellen Zustand durch die aktuellen Sensorinformationen darzustellen. Diese Zustandsrepräsentation wurde zum Beispiel in [HOHK06] [SGD12b] [OHF02] [NYWI08] gewählt. Eine andere Möglichkeit ist die Darstellung des aktuellen Zustand durch die Lage des Fahrzeugs in einem zweidimensionalen Koordinatensystem (x-Koordinate, y-Koordinate) und durch den Winkel des Fahrzeugs. Beide Möglichkeiten werden hier kurz erläutert.

4.2.1 Beschreibung des Zustands durch „rohe“ Sensordaten

Bei der Zustandsrepräsentation durch Sensordaten werden die von den Sensoren gemessenen Entfernungen ohne eine weitere Verarbeitung dazu verwendet, dem Agenten seine aktuelle



(a) Zustand 1

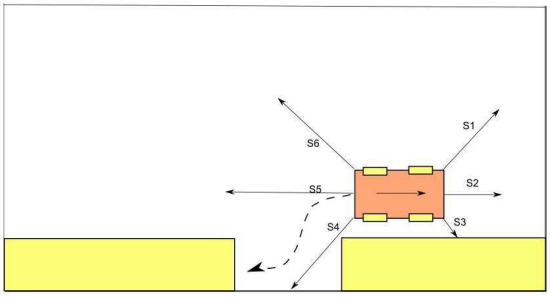


(b) Zustand 2

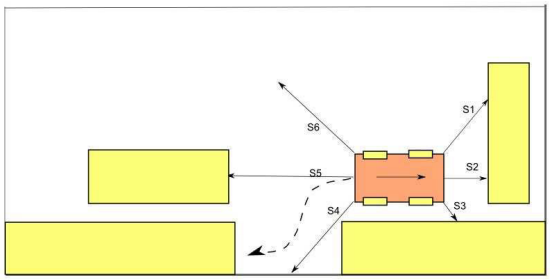
Abbildung 4.4: Verschiedene Zustände

Situation darzustellen. Dies ist in Abbildung 4.3 abgebildet. Der Vorteil dieser Zustandsrepräsentation liegt darin, dass die gemessenen Werte nicht aufwändig weiter vorverarbeitet werden müssen, sondern direkt an das Modul geleitet werden können, welches für die Funktionsapproximation zuständig ist.

Es sprechen jedoch mehrere Nachteile gegen eine solche Zustandsrepräsentation. Zum einen sind auf diese Weise dargestellte Zustände nicht in jedem Fall eindeutig. Ein Beispiel hierfür ist in Abbildung 4.4 zu sehen. Dort sind zwei unterschiedliche Situationen gezeigt, in denen sich das Fahrzeug befindet. In Abbildung 4.4 (a) – Zustand 1 – befindet es sich hinter der Parklücke und kann aus dieser Situation direkt damit beginnen, rückwärts in die Parklücke zu manövrieren. In Abbildung 4.4 (b) –Zustand 2 – hingegen befindet sich das Fahrzeug noch vor der Parklücke. Aus dieser Position heraus, kann es nicht rückwärts in die Parklücke manövrieren, sondern müsste hierfür zunächst in einen Zustand gelangen, der Zustand 1 ähnlich ist. Wird der aktuelle Zustand jedoch nur durch Sensoren beschrieben, kann der Agent diese beiden unterschiedlichen Situationen nicht voneinander unterscheiden. Sie werden als derselbe Zustand wahrgenommen.



(a) Zustand 1



(b) Zustand 2

Abbildung 4.5: Unterschiedliche Darstellung des gleichen Zustands

Bei dem bereits vorgestellten Projekt SEVA3D [HOHK06] wird diese Problematik gelöst, indem der Einparkvorgang in mehrere Schritte unterteilt wurde (wie z.B. Parklücke suchen, neben der Parklücke ausrichten, etc.). Dem Agenten wurde neben den Sensordaten noch der aktuelle Einparkschritt mitgeteilt. Wenn der menschliche Trainer zur Generierung der Trainingsdaten jedoch nur das Fahrzeug mehrfach einparken soll, ohne dabei anzugeben, in welchem Einparkschritt er sich gerade befindet, müsste man eine Möglichkeit finden, den Trainingsdaten diesen automatisch als weiteres Attribut hinzuzufügen.

Ein weiterer Nachtteil der Zustandsrepräsentation durch Sensordaten ist in Abbildung 4.5 gezeigt. In Abbildung 4.5(a) – Zustand 1 – befindet sich der Agent hinter der Parklücke und kann direkt mit dem Einparkvorgang beginnen, indem er sich rückwärts in diese hinein manövriert.

Abbildung 4.5(b) – Zustand 2 – zeigt quasi die gleiche Situation. Auch hier kann der Agent auf die gleiche Weise in die Parklücke manövrieren. Die beiden zusätzlichen Hindernisse in dieser Situation haben eigentlich keinerlei Einfluss auf den Einparkvorgang, da sie sich nicht auf der geplanten Bahn befinden oder den Raum, der dem Agenten zum Einparken zur Verfügung steht, relevant beeinflussen. Es handelt sich quasi in beiden Situationen um das gleiche Problem.

Trotzdem werden diese beiden Situationen einem Agenten, der seinen aktuellen Zustand durch Sensordaten dargestellt bekommt, als unterschiedliche Zustände präsentiert, da Sensor 1, Sensor 2 und Sensor 5 wegen der beiden – eigentlich für das Problem irrelevanten – Hindernisse in Zustand 2 andere Werte liefern als in Zustand 1. Auf diese Weise lassen sich sehr viele Situationen generieren, bei denen die Situation quasi identisch ist, der Zustand von den Entfernungssensoren jedoch stark unterschiedlich dargestellt wird.

Dies bedeutet, dass bei der Zustandsrepräsentation durch Sensordaten die gleiche Situation durch sehr viele unterschiedliche Zustände beschrieben werden kann. Der Zustandsraum wird folglich sehr groß, was es nahezu unmöglich macht, die anzunähernde Funktion in allen markanten Stellen durch Beispiele zu beschreiben. Es werden sehr viele Trainingsbeispiele benötigt, ohne dass man sich sicher sein kann, dass der Zustandsraum in jedem Bereich ausreichend abgetastet wurde, um eine adäquate Approximation der gewünschten Funktion zu gewährleisten.

Eine Möglichkeit, dieses Problem zu lösen, wäre es, eine Sensorik zu verwenden, die speziell an das zu lösende Problem angepasst ist. In Abbildung 4.5 soll das Fahrzeug in eine Parklücke einparken, die sich parallel zur Fahrbahn befindet. Für dieses Problem liefern die Sensoren S_1, S_2 und S_6 in den meisten Fällen keine relevanten Informationen. Es wäre daher möglich, den Zustandsraum zu verkleinern, indem man auf deren Werte verzichten und einen Zustand lediglich durch die gemessenen Entfernungen von S_3, S_4 und S_5 beschreiben würde. Weiterhin könnte man eventuell auch die Ausrichtung der verbliebenen Sensoren ändern, um so relevantere Werte für die Zustandsrepräsentation zu erhalten. Trotz allem bleibt der Zustandsraum auch bei einer an das Problem angepassten Sensorik sehr groß.

Möchte man den Zustand durch die von den Sensoren gemessenen Entfernungen beschreiben, besteht weiterhin das Problem, dass man herausfinden muss, wie viele Sensoren benötigt werden und wie diese am Fahrzeug angebracht werden sollten, um den Agenten alle benötigten Informationen mitzuteilen. Der durch Sensorwerte beschriebene Zustand muss genügend Informationen enthalten, um daraus eine angebrachte Aktion ableiten zu können. Es sollten dem Agenten jedoch auch nicht mehr Informationen als benötigt zur Verfügung gestellt werden, da dies das Lernen verlangsamen würde.

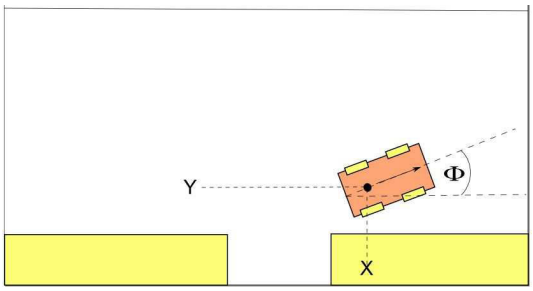


Abbildung 4.6: Repräsentation des Zustands durch Fahrzeugposition und Fahrzeugwinkel

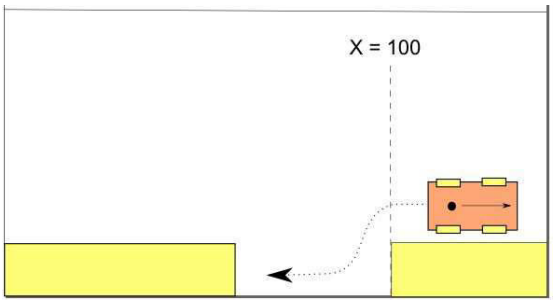
4.2.2 Beschreibung des Zustands durch die Position des Fahrzeugs und den Fahrzeugwinkel

Eine andere Möglichkeit ist es, den aktuellen Zustand durch die Position (x-Wert und y-Wert im Koordinatensystem) und den Fahrzeugwinkel ϕ zu beschreiben (siehe Abbildung 4.6). Diese Zustandsbeschreibung hat gegenüber der im vorherigen Abschnitt vorgestellten Zustandsrepräsentation den Vorteil, dass sie in jedem Fall eindeutig ist. Weiterhin gehen nur drei Werte in den Zustand ein, wodurch der Agent eine sehr reine Information – also nur die Informationen, die er wirklich benötigt – erhält. Dies wirkt sich im Allgemeinen positiv auf die Dauer des Trainings aus, da kein überflüssiger Input ausgewertet werden muss.

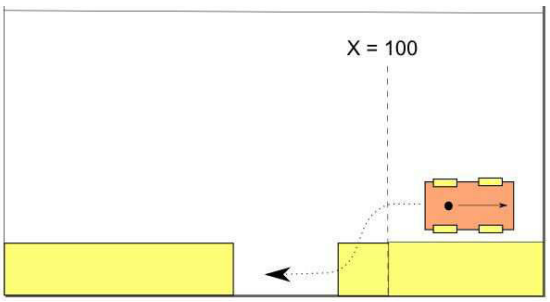
Problematisch bei dieser Darstellung des Zustands ist, dass Hindernisse oder Fahrzeuge, die die Parklücke begrenzen, nicht berücksichtigt werden. Ein mögliches Fehlerszenario, das mit diesem Problem verbunden ist, ist in Abbildung 4.7 zu sehen. In diesem Szenario ist die Parklücke, mit der die Trainingsdaten generiert wurden, größer als die Parklücke, in die der Agent nun selbstständig einparken soll.

Der menschliche Trainer hatte während der Trainingsdatengenerierung die Strategie, immer rückwärts bis zur Position $X = 100$ zu fahren und dort voll einzuschlagen. Lernt der Agent nun diese Strategie und wendet sie auf eine Parklücke an, die kleiner als die Parklücke ist, auf die die Strategie des Trainers angewendet wurde, fährt der Agent in ein Hindernis.

Dieses Problem lässt sich lösen, indem zur Generierung der Trainingsdaten die kleinst mögliche Parklücke verwendet wird, in die das Fahrzeug einparken kann. Ist die Parklücke größer als diese kleinste Parklücke, kann der Parkvorgang trotzdem so durchgeführt werden, als hätte die größere Lücke die Größe der kleineren. Dies ist in Abbildung 4.8 zu sehen.



(a) Trainingsdatengenerierung



(b) Anwendung des erlernten

Abbildung 4.7: Problem der Zustandsdarstellung durch Fahrzeugposition und Fahrzeugwinkel

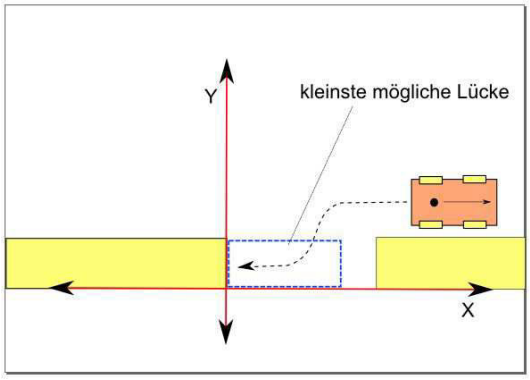


Abbildung 4.8: Koordinatensystem und kleinste mögliche Parklücke

Abbildung 4.9: Mögliche zu erlernende Abbildung

Weiterhin ist in dieser Abbildung zu sehen, dass sich der Koordinatenursprung immer in der linken unteren Ecke der Parklücke befindet, in die eingeparkt werden soll. Auf diese Weise wird die aktuelle Fahrzeugposition immer relativ zur Parklücke angegeben. Auf diese Weise wird der Zustandsraum kleiner gehalten und es ist wahrscheinlicher, dass der Zustandsraum ausreichend abgetastet wird.

Durch diese beiden Modifikationen bei der Darstellung des Zustands ist es nicht mehr notwendig, dem Agenten mitzuteilen, an welchen Positionen sich die beiden Hindernisse befinden, die die Parklücke begrenzen. Die vom Trainer in den Trainingsdaten beschriebene Trajektorie leitet das Fahrzeug in jedem Fall in die Lücke – vorausgesetzt diese ist nicht kleiner als die Parklücke, die als die kleinst möglichste Parklücke angenommen wurde.

4.3 Generierung der Lerndaten

Da der Agent (das Fahrzeug) in dieser Arbeit durch überwachtes Lernen das Einparken in eine Parklücke lernen sollte, müssen in einem ersten Schritt zunächst Trainingsdaten generiert werden, aus denen er das gewünschte Verhalten erlernen kann.
Während des Einparkvorgangs nimmt der Agent verschiedene Zustände ein, die z.B. durch seine aktuelle Position und/oder Abstände zu Hindernissen beschrieben werden können. Sein Ziel ist es nun, eine Abbildung von seinem aktuellen Zustand auf eine dem Zustand angemessene Aktion zu erlernen. In Abbildung 4.9 ist eine mögliche zu erlernende Abbildung gezeigt. In diesem Beispiel wird der aktuelle Zustand des Agenten durch seine aktuelle Position in einem Koordinatensystem (x,y) und durch den aktuellen Winkel des Fahrzeugs (ϕ) beschrieben. Diese Eingabe soll nun auf eine Aktion abgebildet werden, die den Agenten seinem Ziel, in eine Parklücke einzuparken, ein Stück näher bringt. Diese Aktion wird in dem in Abbildung 4.9 beschriebenen Beispiel durch die Richtung, in die er fahren soll (vorwärts/rückwärts), und einen Lenkwinkel (α), den er einschlagen soll, beschrieben. Die Approximation dieser Abbildung ist Ziel des Lernvorgangs.

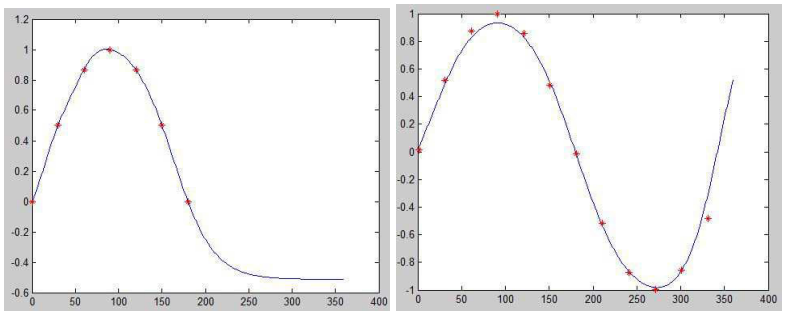
Damit der Agent diese Abbildung erlernen kann, müssen ihm Trainingsbeispiele vorgelegt werden. Diese bestehen aus Eingaben $\vec{X}$ und dazugehörigen Ausgaben $\vec{Y}$. Diese müssen zunächst von einem menschlichen Trainer erzeugt werden. Hierzu kann das Fahrzeug in der Simulationsumgebung eingeparkt werden. Dabei können verschiedenen Daten (z.B. Sensorwerte, Fahrzeugposition und Lenkwinkel) während des Einparkvorgangs gespeichert werden.

Zum Erzeugen der Trainingsdaten startet das Fahrzeug an einer definierten Startposition und wird im Anschluss von dem Trainer in die Parklücke navigiert. Dabei werden in regelmäßigen Abständen der aktuelle Zustand des Fahrzeugs und die dazugehörige vom Trainer ausgeführte Aktion aufgezeichnet. Nach mehreren Einparkvorgängen liegen dann genügend Trainingsdaten vor, um den Agenten mit diesen zu trainieren.

Wichtig beim Generieren der Trainingsdaten ist, dass der Zustandsraum ausreichend abgetastet wird, damit der Agent später aus aktuell vorliegenden Daten eine gute Aktion herleiten kann. Ausreichend abgetastet bedeutet dabei nicht, dass jeder Punkt im Zustandsraum in den Lerndaten vorhanden sein muss. Dies würde bedeuten, dass der Agent zu jedem möglichen Zustand die dazugehörige Aktion präsentiert bekäme und später bei der Ausführung nur noch nachschlagen müsste, welche Aktion er im momentanen Zustand wählen muss.

Vielmehr muss der Zustandsraum so abgetastet sein, dass durch die Trainingsdaten markante Punkte der zu approximierenden Funktion festgelegt werden. Durch Generalisierung kann der Agent dann die dazwischenliegenden Werte herleiten. Außerhalb des durch die Trainingsdaten beschriebenen Bereichs ist eine Generalisierung nicht möglich, da dort keine Beispieldaten vorliegen, an denen festgemacht werden kann, welche Form die anzunähernde Funktion dort annimmt.

Bezogen auf das Einparkproblem bedeutet dies, dass das System aus Abbildung 4.9 zu gegebenen x,y,ϕ nur dann eine adäquate Aktion herleiten kann, wenn diese Werte in einem Bereich liegen, für den das System Trainingsdaten vorliegen hatte. Wurden dem System z.B. während des Trainings nur Trainingsdaten aus dem Bereich $0 \leq x \leq 100$, $0 \leq y \leq 100$, $-45 \leq \phi \leq 45$ präsentiert, so wird es nach dem Training den Funktionswert für die Eingabe $x = 20, y = 40, \phi = 20$ mit einer hohen Wahrscheinlichkeit gut approximieren können. Bei einer Eingabe von $x = 150, y = 200, \phi = 45$ fehlen jedoch die Informationen, welchen Wert die anzunähernde Funktion hier annehmen könnte, da diese Eingabe weit außerhalb des



(a) Schlechte Abtastung des Eingaberaums (b) Gute Abtastung des Eingaberaums

Abbildung 4.10: Approximation der Sinus-Funktion durch ein Neuronales Netz

trainierten Bereichs liegt.

Ein weiteres konkretes Beispiel ist in Abbildung 4.10 gezeigt. Hier sollte die Sinus-Funktion im Bereich $[0;2\pi]$ mit Hilfe eines Neuronalen Netzes angenähert werden. Das Ergebnis der Approximation ist in beiden Beispielen blau eingezeichnet, die Trainingsdaten rot.

Dort ist zu erkennen, dass die Funktion von dem Neuronalen Netz, welches mit Trainingsdaten aus dem gesamten Eingaberaum ($0 \leq x \leq 2\pi$) trainiert wurde, im gesamten Bereich ohne großen Fehler approximiert wird. Dem Neuronalen Netz, welchem nur Daten aus ($0 \leq x \leq \pi$) zum Trainieren gegeben wurden, fehlen im Bereich ($x > \pi$) Trainingsdaten, an denen es den dortigen Verlauf der Funktion festlegen kann. Die dort nach dem Training berechneten Werte weichen sehr stark vom eigentlichen Verlauf der Sinus-Funktion ab. Die Bereiche zwischen den in den Trainingsdaten präsentierten Punkten hingegen konnten beide Netze ausreichend generalisieren.

Aus diesem Grund ist es wichtig, dass der Eingaberaum während des Generierung der Daten möglichst vollständig besucht wird. Man kann nicht erwarten, dass der Agent aus einer vollkommen neuen Situation heraus eine geeignete Trajektorie in die Parklücke findet.

4.4 Funktionsapproximation

Im vorherigen Abschnitt wurde beschrieben, auf welche Weise die Trainingsdaten generiert wurden. Dort wurde ebenfalls gesagt, dass das Erlernen einer dem Zustand angemessenen

Aktion als das Annähern einer Funktion aufgefasst werden kann. Für diese Aufgabe wurden in dieser Arbeit monolithische Neuronale Netze und Modulare Neuronale Netze verwendet. Es sind zahlreiche Methoden zur Annäherung einer Funktion vorhanden (wie z.B. die Approximation mit Polynomen) und neben Neuronalen Netzen als weitere Möglichkeiten zur Funktionsapproximation denkbar. Durch die Verwendung von Neuronalen Netzen wurde jedoch ein Verfahren gewählt, welches durch eine Änderung der Neuronenanzahl in der verdeckten Schicht leicht an eine verändernde Komplexität der zu erlernenden Aufgabe angepasst werden kann.

In diesem Abschnitt soll der allgemeine Aufbau der verschiedenen Approximationsmodule erläutert werden.

4.4.1 Funktionsapproximation durch vorwärtsgekoppelte Neuronale Netze

Eine naheliegende Möglichkeit zur Annäherung der gesuchten Funktionen sind vorwärtsgekoppelte Neuronale Netze. Neben der Entscheidung, wie die einzelnen Neuronen untereinander verbunden sind, kann man beim Entwurf eines Neuronalen Netzes dessen Topologie durch zahlreiche weitere Parameter beeinflussen. Einige von diesen Parametern sind z.B. die Anzahl der verdeckten Schichten, die Anzahl der Neuronen pro verdeckter Schicht und die Aktivierungsfunktion der Neuronen.

Dabei gibt es keine allgemeingültigen Regeln dazu, wie viele verdeckte Schichten ein Netz besitzen und wie viele Neuronen sich in diesen befinden sollten. Generell kann man sagen, dass man die Zahl der Neuronen in der verdeckten Schicht so wählen sollte, dass das resultierende Netz am besten gerade flexibel genug ist, um die Funktion approximieren zu können, jedoch nicht flexibler. Wählt man zu wenige Neuronen, kann die Aufgabe eventuell nicht ausreichend gut gelernt werden, während das Netz bei zu vielen Neuronen lediglich als Speicher fungiert: Es kann die gelernten Trainingsdaten zwar mit einem sehr geringen Fehler wiedergeben, besitzt jedoch eine schlechte Generalisierungsfähigkeit.

Um eine angemessene Netzgröße herauszufinden, bietet es sich an, mit mehreren Varianten zu experimentieren und aus diesen die Beste zu wählen oder gute Ansätze zu verfeinern.

Während dieser Arbeit wurden verschiedene Versuche durchgeführt, in denen untersucht wurde, auf welche Weise der Agent am besten die Aufgabe löst, in eine Parklücke zu navigieren. Hierzu wurden verschiedene Netztopologien gewählt, die im Detail bei den dazugehörigen

Versuchen beschrieben werden. Ihnen gemeinsam sind jedoch verschiedene Verfahren, die angewendet wurden, um den Fehler bei der Funktionsapproximation so gering wie möglich zu halten und gleichzeitig eine möglichst gute Generalisierungsfähigkeit zu erhalten. Außerdem sollten sie dazu beitragen, die Trainingszeit, welche das Netz benötigt, um die Trainingsdaten zu erlernen, zu verringern.

Skalierung der Eingangswerte

Ein unterschiedlicher Wertebereich der Eingabegrößen führt bei Neuronalen Netzen im Allgemeinen zu unterschiedlichen Lerneinflüssen [Bit02]. Ein Beispiel hierfür wären die beiden Eingaben $x_1 \in [500, 800]$ und $x_2 \in [2, 5]$ In diesem Fall dominiert x_1 die andere Einagabe x_2, da die Multiplikation mit dem jeweiligen Eingangsgewicht nicht ausreicht, um diese beiden stark unterschiedlichen Wertebereiche anzugleichen. Weiterhin kann das Training durch eine solche Vorverarbeitung der Eingangsdaten effizienter gemacht werden [Mat10a].

Zur Skalierung der Eingangswerte stellt Matlab verschiedene Funktionen zur Verfügung. In dieser Arbeit wurde vor dem Training die Funktion „mapminmax“ auf die Trainingsdaten angewandt. Dies entspricht einer linearen Transformation. Mit Hilfe der Minimal- und Maximalwerte in den Eingangsdaten werden die Daten auf einen Wertebereich von [-1,1] umgerechnet. Das Netz wird nun mit diesen normierten Daten trainiert. Nutzt man ein Neuronales Netz, das mit auf diese Weise vorverarbeiteten Daten trainiert wurde, müssen sämtliche neuen Eingaben ebenfalls mit den während des Trainings zur Skalierung benutzen Minimal-und Maximalwerten vorverarbeitet werden.

Early Stopping

Um Overfitting – die Überanpassung der Gewichte an die Lerndaten – zu vermeiden, verwendet Matlab während des Trainings Early Stopping. Hierzu wird die Menge der Trainingsdaten in 3 Teilmengen aufgeteilt:

Trainingsmenge: Mit Hilfe der Daten aus dieser Teilmenge wird das Netz trainiert. Mit ihnen wird der Gradient und die Anpassung der Verbindungsgewichte berechnet.

Validationsmenge: Die Daten aus dieser Teilmenge werden nicht dazu verwendet, um die Verbindungsgewichte anzupassen. Während des Trainings wird der Fehler, den das Netz bei diesen Daten macht, beobachtet. Normalerweise sinkt der Fehler bei dieser Menge anfangs ebenso wie der Fehler der Daten aus der Trainingsmenge. Sobald Overfitting beginnt, steigt jedoch der Fehler bei den Daten der Validationsmenge, während der

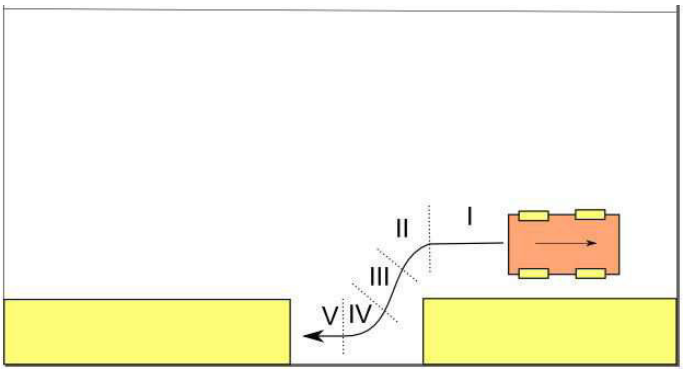


Abbildung 4.11: Phasen eines Einparkvorgangs

Fehler bei der Trainingsmenge weiter sinkt. Wenn der Fehler in der Validationsmenge für eine festgelegte Anzahl von Iterationen steigt, wird das Training durch Early Stopping automatisch abgebrochen, um eine Überanpassung des Netzes zu vermeiden.

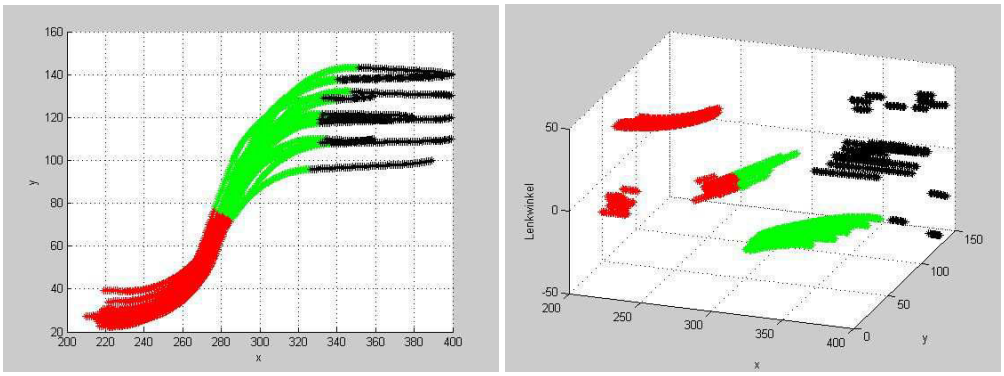
Testmenge: Diese Teilmenge wird nicht während des Training benutzt und kann dazu genutzt werden, um verschiedene Modelle zu vergleichen.

4.4.2 Modulare Neuronale Netze

Betrachtet man einen konkreten Einparkvorgang, so zeichnet dieser sich in der Regel dadurch aus, dass auf engem Raum relativ stark gelenkt wird, um das Fahrzeug in die Parklücke zu navigieren. Wenn man den Einparkvorgang als eine Funktion darstellen möchte, bei der der aktuelle Zustand auf einen Lenkwinkel α abgebildet wird, so wird diese normalerweise nicht glatt sein, sondern ihren Funktionswert zwischen zwei benachbarten Zuständen relativ stark ändern.

Ein Beispiel hierfür ist in Abbildung 4.11 zu sehen. In diesem Beispiel soll das Fahrzeug rückwärts in eine Parklücke einparken, die sich parallel zur Fahrbahn befindet. Während das Fahrzeug in Phase I nahezu gerade zurück fährt ($\alpha = 0$), wechselt der Lenkwinkel beim Übergang zu Phase II abrupt zu $\alpha = -45$ (volleinschlag rechts). Ein solches Verhalten kommt in mehr oder weniger ausgeprägter Form an allen Übergangen von einer Phase zu einer anderen vor.

Man kann also davon ausgehen, dass die anzunähernde Funktion nicht glatt ist, sondern abrupte Änderungen im Wertebereich aufweist. Aus diesem Grund wurden in dieser Arbeit neben monolithischen Netzen auch Modulare Neuronale Netze zur Funktionsapproximation



(a) Zweidimensionale Darstellung (b) Dreidimensionale Darstellung

Abbildung 4.12: Eine mögliche Einteilung der Trainingsdaten in Cluster

genutzt, die Funktionen mit den genannten Eigenschaften im Allgemeinen besser approximieren [JJ93] [FH96].

Es wurde der von Bennani [Ben95] beschriebene halbautomatische Ansatz gewählt, bei dem die Daten in einem ersten Schritt zu verschiedenen Clustern zusammengefasst werden. Ein Beispiel für eine mögliche Clusterung der Trainingsdaten ist in Abbildung 4.12 zu sehen. Beide Abbildung zeigen den gleichen Sachverhalt, in Abbildung 4.12(a) wird die Darstellung jedoch auf zwei Dimensionen beschränkt, während in Abbildung 4.12(b) die gleichen Daten mit den dazugehörigen Lenkwinkeln dargestellt werden.

Nach der Clusterung werden die Expertennetze trainiert. Jedes Expertennetz wird dabei mit den Daten eines Clusters trainiert und soll nach dem Training die zu erlernende Funktion in diesem Bereich approximieren. Die Trainingsdaten werden also nach Regionen getrennt. Hierdurch wird der Wertebereich, den ein einzelnes Expertennetz erlernen muss, reduziert, was zu einem besseren Training gegenüber den monolithischen Netzen führt. Um nun zu einem konkreten Zustand des Agenten das Expertennetz auszuwählen, das für diesen Zustand zuständig ist, wird ein weiteres Netz benötigt. Dies ist das Weichennetz. Es wird mit den gesamten Trainingsdaten trainiert und lernt, diese zu klassifizieren. Die Klassen, sind die im vorherigen Schritt erzeugten Cluster. Es ordnet seine Eingaben folglich einem der Cluster zu, wodurch im nächsten Schritt das Expertennetz bestimmt werden kann, welches für dieses Cluster zuständig ist. Dieses Cluster wird dann auch den Funktionswert für die aktuelle Eingabe berechnen. Wie eine Architektur eines solchen modularen Neuronalen Netzes aussehen könnte, ist noch einmal in Abbildung 4.13 gezeigt. Die Eingabe gelangt zunächst an das Weichennetz, welches

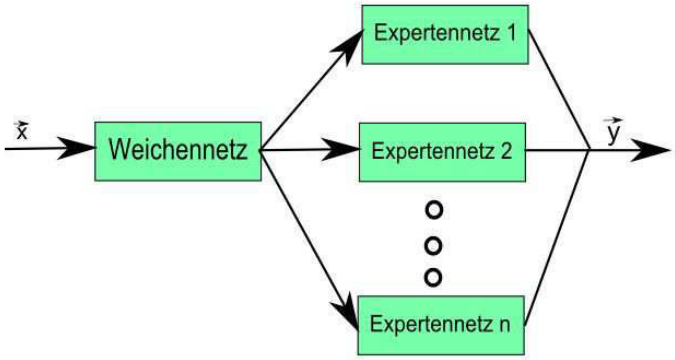


Abbildung 4.13: Beispielarchitektur eines modularen Neuronalen Netzes

dieser Eingabe ein Cluster und damit verbunden das Expertennetz zuweist, das für dieses Cluster zuständig ist. Das ausgewählte Expertennetz berechnet nun für die aktuelle Eingabe eine Ausgabe.

5 Vergleich der umgesetzten Verfahren

5.1 Gewählte Netztopologien

Generell wurden zur Funktionsapproximation vorwärtsgekoppelte Netze mit einer Eingabe-, einer Ausgabe- und einer verdeckten Schicht gewählt. Bei den Versuchen haben sich Netze mit nur einer verdeckte Schicht als geeigneter herausgestellt, als solche mit mehreren. Als Aktivierungsfunktion der Neuronen der verdeckten Schicht wurde die Tan-Sigmoid Funktion gewählt. Die Aktivierungsfunktion der Neuronen der Ausgabeschicht ist linear, also im Gegensatz zur Tan-Sigmoid Funktion nicht auf den Wertebereich [-1,1] beschränkt. Diese Topologie wird als besonders geeignet zur Funktionsapproximation beschrieben [Mat10a].

Der Aufbau eines solchen Netzes ist ein Abbildung 5.1 gezeigt.

Netze, die zu Klassifizierungszwecken trainiert wurden, sind ebenfalls vorwärtsgekoppelte Netze und bestehen ebenfalls aus Eingabe-, Ausgabe- und verdeckter Schicht. Bei ihnen wurde jedoch sowohl für die Aktivierungsfunktion der Neuronen der verdeckten Schicht, als auch für die Neuronen der Ausgabeschicht die Tan-Sigmiode Funktion gewählt.

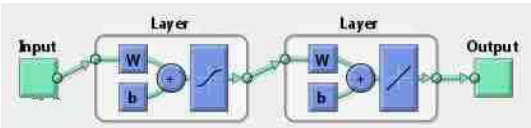


Abbildung 5.1: Aufbau eines Netzes zur Funktionsapproximation

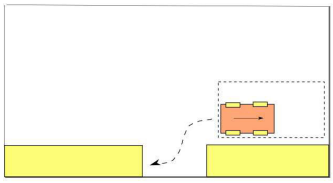
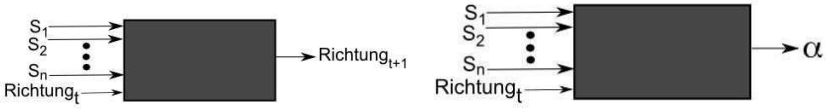


Abbildung 5.2: Rückwärts einparken



(a) Neuronales Netz zur Berechnung der Richtung als Blackbox

(b) Neuronales Netz zur Berechnung des Lenkwinkels als Blackbox

Abbildung 5.3: Eingänge aus Ausgänge der Neuronalen Netze in Szenario 1

5.2 Szenario 1

5.2.1 Zielstellung und Randbedingungen

Bei diesem Szenario galt folgende Aufgabenstellung:

Zielstellung:

- Der Agent soll durch Rangieren in die Parklücke einparken.
- Zu einem Zustand (siehe Randbedingungen) sollen eine Richtung und der Lenkwinkel als Ausgabe erlernt werden.

Randbedingungen:

- Der Agent befindet sich hinter der Parklücke.
- Die Startposition des Agenten kann in x- und y- Richtung variieren. Ebenso der Winkel, in dem sich das Fahrzeug anfangs befindet.
- Der aktuelle Zustand wird durch Werte der Entfernungssensoren beschrieben.

Die Startposition des Agenten ist in Abbildung 5.2 zu sehen.

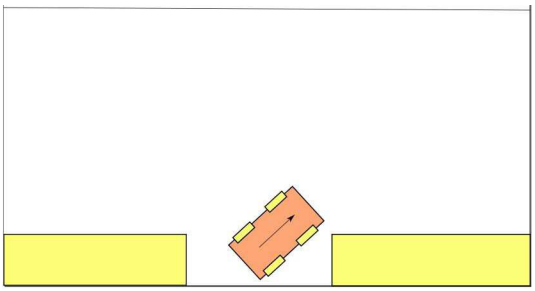


Abbildung 5.4: Zustand ohne Richtung

5.2.2 Funktionsapproximation

Bei diesem Versuch wurden von Entfernungssensoren gemessene Werte zur Zustandsrepräsentation gewählt. Aus diesen sollte der zu dieser Situation passende Lenkwinkel und die passende Richtung erlernt werden. Hierzu wurden zwei Netze verwendet. Eines war für die Berechnung des Lenkwinkels zuständig, während das andere die Richtung berechnet hat. Dies ist in Abbildung 5.3 zu sehen.

Die Netze sind folgendermaßen Aufgebaut:

Eingänge:

1. Pro Entfernungssensor (S_1 bis S_n) einen Eingang.
2. Die aktuelle Richtung (vorwärts/rückwärts), in die sich der Agent bewegt. Diese Eingabe ist für das Rangieren in der Parklücke notwendig. Ein Beispiel ist in Abbildung 5.4 zu sehen. Durch die Sensorinformationen kann die Position des Fahrzeugs bestimmt werden. Fehlen jedoch die Informationen über die aktuelle Richtung, wäre in der abgebildeten Situation unklar, wohin sich das Fahrzeug im Moment bewegt: Es könnte gerade nach vorne fahren, um anschließend wieder zurück zu navigieren. Es könnte sich jedoch auch bereits auf dem Weg zurück in die Parklücke befinden und solange rückwärts fahren, bis er auf ein Hindernis stößt, welches ihn dann dazu zwingt, seine Richtung zu ändern.

Ausgang: Jedes Netz besitzt einen Ausgang, das eine Netz für die neue Richtung, das andere für den Lenkwinkel α.

Anzahl Neuronen in der verdeckten Schicht: 10-15.

5.2.3 Ergebnisse

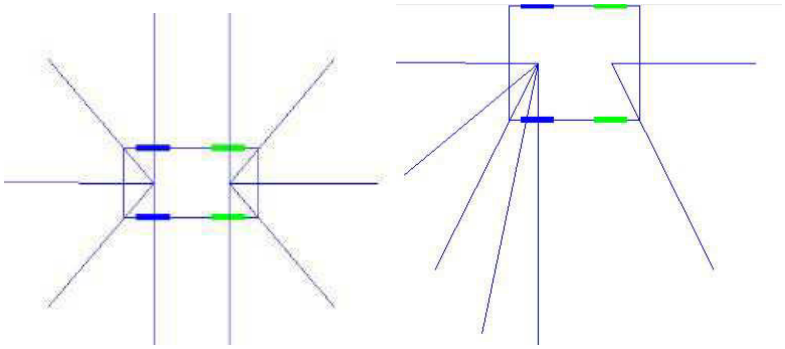
Wurden die Trainingsdaten so generiert, dass das Fahrzeug immer von derselben Startposition gestartet ist und dann mit leichten Variationen in den einzelnen Einparkvorgängen in die Parklücke manövriert wurde, so fährt es auch nach dem Lernvorgang in der Anwendungsphase sicher und ohne mit den die Parklücke begrenzenden Hindernissen zu kollidieren in die Parklücke (sofern die Startposition mit der trainierten Startposition übereinstimmt).

Auch wenn verschieden große Parklücken trainiert werden, die trainierte Startposition jedoch immer dieselbe bleibt, ist der Agent in der Lage, nach dem Lernvorgang von dieser Position aus korrekt in die unterschiedlichen Parklücken zu navigieren. Auch ein Einparken in Parklücken, deren Größe zwischen den trainierten Lücken liegt, ist möglich.

Wird der Einparkvorgang jedoch mit variierenden Startpositionen trainiert, so sinkt die Leistung des Agenten. Häufig entstehen Fehler zu Beginn des Einparkvorgangs. Ein häufig beobachtetes Verhalten ist die Falschberechnung der Richtung, sodass der Agent nach vorne von der Parklücke wegfährt, anstatt auf diese hin zu navigieren. Ein Fehler, der ebenfalls regelmäßig beobachtet wurde, ist dass der Agent anfangs in die falsche Richtung gelenkt und sich dadurch von der Parklücke entfernt hat. Bei beiden Fehlern entfernt sich der Agent von der Parklücke und gerät in Bereiche, die nicht trainiert wurden. Da der Lenkwinkel und die Richtung in diesen Bereichen nicht zuverlässig approximiert werden, bewegt sich das Fahrzeug aus einem solchen Bereich nicht mehr auf eine Trajektorie zurück, die in die Parklücke hinein führt.

Ein weiteres Verhalten, das regelmäßig beobachtet werden konnte, ist, dass der Agent außerhalb der Parklücke mit einer hohen Frequenz seine Richtung wechselt und sich folglich nicht aus diesem Zustandsbereich bewegt.

Es wurde auch untersucht, wie sich unterschiedlich an dem Fahrzeug angebrachte Sensoren auf die Leistung des Agenten auswirken. In Abbildung 5.5 sind zwei Varianten gezeigt, wie Sensoren am Fahrzeug angebracht werden könnten. Abbildung 5.5(b) zeigt dabei eine Ausrichtung der Sensoren, die speziell an das Problem des Einparkens in eine Parklücke, die sich zur rechten Seite des Fahrzeugs befindet, angepasst ist. Ein Agent, der mit einer solchen Sensorik arbeitete, hatte augenscheinlich eine höhere Erfolgsrate als einer mit einer allgemein ausgelegten Sensorik.



(a) Allgemeine Sensorik (b) Problemangepasste Sensorik

Abbildung 5.5: Unterschiedliche Sensorik

5.3 Szenario 2

5.3.1 Zielstellung und Randbedingungen

Bei diesem Szenario ist die Zielstellung identisch mit der Zielstellung von Szenario 1.

Zielstellung:

- Der Agent soll mit Rangieren in die Parklücke einparken.
- Zu einem Zustand (siehe Randbedingungen) sollen eine Richtung und der Lenkwinkel als Ausgabe erlernt werden.

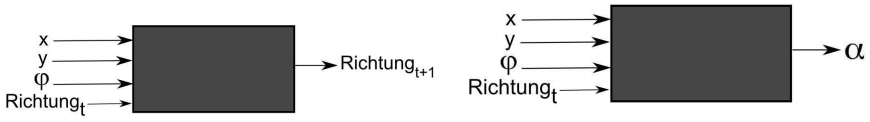
Bei den Randbedingungen hat sich jedoch geändert, dass der Zustand des Agenten nicht mehr durch die von den Sensoren gemessenen Entfernungen, sondern durch die Position des Fahrzeugs und den Fahrzeugwinkel beschrieben wird.

Randbedingungen:

- Der Agent befindet sich hinter der Parklücke.
- Die Startposition des Agenten kann in x- und y- Richtung variieren. Ebenso der Winkel, in dem sich das Fahrzeug anfangs befindet.
- Der aktuelle Zustand wird durch die Fahrzeugposition (x-Koordinate, y-Koordinate) und den Fahrzeugwinkel ϕ beschrieben.

5.3.2 Funktionsapproximation

Bei diesem Versuch sollte zu einem Zustand eine adäquate Richtung und ein adäquater Lenkwinkel erlernt werden. Wie in Abbildung 5.6 zu sehen ist, wurde hierbei der Zustand jedoch



(a) Neuronales Netz zur Berechnung der Richtung als Blackbox

(b) Neuronales Netz zur Berechnung des Lenkwinkels als Blackbox

Abbildung 5.6: Eingänge aus Ausgänge der Neuronalen Netze in Szenario 2

wegen der in Abschnitt 4.2 beschriebenen Vorteile (eindeutig und einfacher als die Beschreibung durch Sensoren) durch die x-Koordinate, die y-Koordinate und den Fahrzeugwinkel ϕ beschrieben. Auch hier wurden wieder zwei Netze verwendet: eines für die Berechnung des Lenkwinkels α, das andere für die Berechnung der Richtung.

Aufbau der Netze:

Eingänge: Jedes Netz besitzt 4 Eingänge

1. x-Koordinate des Agenten
2. y-Koordinate des Agenten
3. Fahrzeugwinkel ϕ
4. Richtung, in die sich der aktuell Agent bewegt (wichtig zum Rangieren).

Ausgang: Jedes Netz besitzt einen Ausgang. Das eine Netz für die Richtung, das andere für den Lenkwinkel α.

Anzahl Neuronen in der verdeckten Schicht: 15.

5.3.3 Ergebnisse

Im Gegensatz zu Szenario 1 ist der Agent in während der Anfangsphase bei allen Einparkvorgängen korrekt in Richtung der Parklücke gefahren. Auch die Lenkvorgänge in dieser Phase wurden korrekt gelernt, sodass das Fahrzeug der gelernten Trajektorie gefolgt ist. In der Parklücke jedoch tritt der in Abbildung 5.7 gezeigte Fehler auf: Der Agent erkennt nicht rechtzeitig, dass er seine Richtung ändern muss und kollidiert mit einer der Parklückenbegrenzungen. Dieser Fehler tritt bei nahezu jedem Einparkversuch auf.

Ein möglicher Grund für dieses Verhalten könnte sein, dass der Zustand die aktuelle Po-

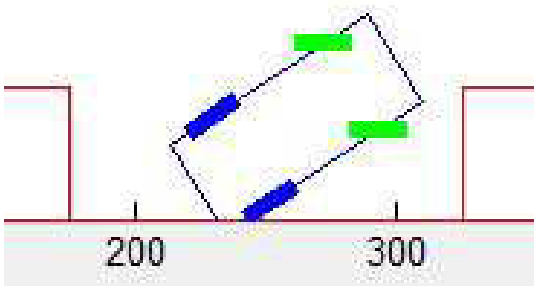


Abbildung 5.7: Zustand ohne Richtung

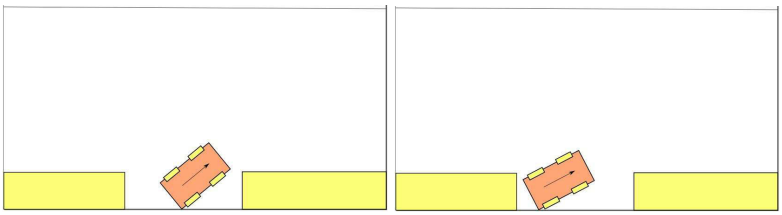


Abbildung 5.8: Beispiel für unterschiedliche Zustände, in denen die Richtung gewechselt werden muss

sition des Agenten beschreibt, dabei jedoch nicht so eindeutig aufzeigt, wann der Agent die Richtung wechseln muss, um nicht gegen ein Hindernis zu fahren. In Abbildung 5.8 wird gezeigt, dass es bei dieser Art der Zustandsdarstellung sehr viele unterschiedliche Zustände gibt, bei denen der Agent die Richtung wechseln muss, während ein Agent, der seinen Zustand durch Sensordaten bestimmt, einfach gelernt haben könnte, dass ein Richtungswechsel nötig ist, sobald eine bestimmte Distanz unterschritten ist.

5.4 Szenario 3

5.4.1 Zielstellung und Randbedingungen

Bei diesem Szenario wurde bei der Zielstellung im Vergleich zu den vorherigen Verfahren geändert, dass der Agent nun nicht mehr durch Rangieren in die Parklücke gelangen, sondern in einem Zug einparken soll. Die Lernaufgabe wurde also vereinfacht. Aus diesem Grund musste zu einem Zustand auch nur noch ein adäquater Lenkwinkel erlernt werden, da die Richtung durch die Aufgabenstellung bereits automatisch gegeben ist. Um in einem Zug rückwärts einzuparken, muss der Agent sich die gesamte Zeit über rückwärts bewegen.

Zielstellung:

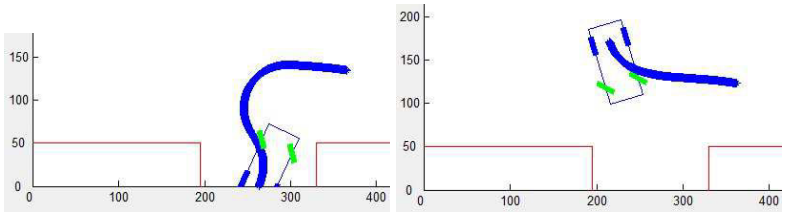
- Der Agent soll in einem Zug (ohne Rangieren) in die Parklücke einparken.
- Zu einem Zustand (siehe Randbedingungen) soll ein Lenkwinkel α als Ausgabe erlernt werden.

Randbedingungen:

- Der Agent befindet sich hinter der Parklücke.
- Die Startposition des Agenten kann in x- und y- Richtung variieren. Ebenso der Winkel, in dem sich das Fahrzeug anfangs befindet.
- Der aktuelle Zustand wird durch die Fahrzeugposition (x-Koordinate, y-Koordinate) und den Fahrzeugwinkel ϕ beschrieben.

5.4.2 Funktionsapproximation

Da bei diesem Szenario auf die Berechnung der Richtung verzichtet wurde, wurde in diesem Versuch nur noch ein vorwärts gerichtetes Neuronales Netz zur Berechnung des Lenkwinkels genutzt.



(a) (b)

Abbildung 5.9: Beobachtetes Verhalten in den Grenzbereichen

Aufbau des Netzes:

Eingänge: Das Netz besitzt 3 Eingänge. Eine Richtung ist nicht mehr notwendig, da keine Richtungswechsel mehr erlernt werden müssen.

1. x-Koordinate des Agenten
2. y-Koordinate des Agenten
3. Fahrzeugwinkel ϕ

Ausgang: Das Netz besitzt einen Ausgang zur Ausgabe des Lenkwinkels α.

Anzahl Neuronen in der verdeckten Schicht: 5

5.4.3 Ergebnisse

Wird nur das Einparken ausgehend von immer derselben Startposition trainiert, so ist das Einparken in die Parklücke nach dem Training problemlos möglich, sofern das Fahrzeug den Einparkvorgang von dieser trainierten Startposition aus beginnt. Variiert man die Startposition jedoch, nimmt auch hier die Leistung des Agenten ab. Nach Augenschein ist die Erfolgsrate des Agenten deutlich höher als bei den vorherigen Versuchen. Trotzdem treten besonders bei Startpositionen, die sich an den Grenzen des erlernten Bereichs befinden, Fehler auf. Diese sind in den Abbildungen 5.9 (a) und 5.9 (b) abgebildet. Weiterhin wurde beobachtet, dass der Agent oftmals keine Position einnimmt, die sich parallel zur unteren Begrenzung der Parklücke befindet, sondern immer schräg zu dieser einparkt.

Automatisierte Einparküberprüfungen, bei denen ein Agent 200 mal von unterschiedlichen, zu-

fällig gewählten Startpositionen innerhalb des trainierten Bereichs gestartet ist, haben ergeben, dass der Agent in 59,5% der Einparkversuche erfolgreich eingeparkt ist[1].

5.5 Szenario 4

5.5.1 Zielstellung

Die Zielstellungen und Randbedingungen sind identisch mit jenen aus dem vorherigen (Szenario 3).

Zielstellung:

- Der Agent soll in einem Zug (ohne Rangieren) in die Parklücke einparken.
- Zu einem Zustand (siehe Randbedingungen) soll ein Lenkwinkel α als Ausgabe erlernt werden.

Randbedingungen:

- Der Agent befindet sich hinter der Parklücke.
- Die Startposition des Agenten kann in x- und y- Richtung variieren. Ebenso der Winkel, in dem sich das Fahrzeug anfangs befindet.
- Der aktuelle Zustand wird durch die Fahrzeugposition (x-Koordinate, y-Koordinate) und den Fahrzeugwinkel ϕ beschrieben.

5.5.2 Funktionsapproximation

Bei diesem Szenario wurden zur Funktionsapproximation modulare Neuronale Netze verwendet. Diese bestanden jeweils aus mehreren Expertennetzen und einem Weichennetz. Der Aufbau des modularen Neuronalen Netzes ist in Abbildung 5.10 zu sehen.

Aleotti beschreibt in seinen Arbeiten zu „robot programming by demonstation“, in denen es darum geht, Trajektorien zu approximieren, dass man Trajektorien z.B. an Punkten, an denen keine Bewegung vorhanden ist, teilen kann, um so Teilaufgaben voneinander zu trennen [AC06]. Der Ansatz, Trajektorien zu teilen wird auch anderen Arbeiten beschrieben. Trajektorien werden geteilt oder (Teil-)Trajektorien zu Clustern zusammengefasst, um diese in

[1] Ein Einparkversuch galt als erfolgreich, wenn der Agent ohne mit einem Hindernis zu kollidieren in die Parklücke navigiert ist und dort mit einer Abweichung von bis zu 5° parallel zur unteren Seite der Parklücke stand.

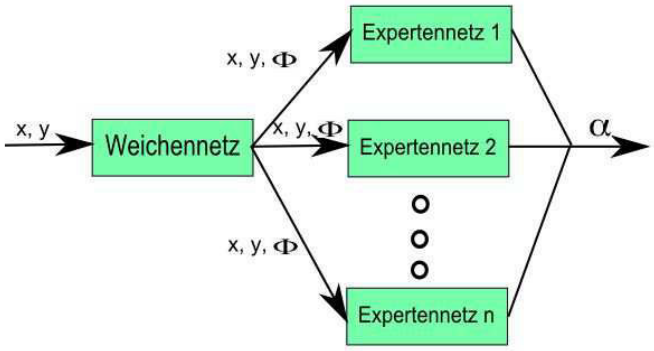


Abbildung 5.10: Aufbau des modularen Neuronalen Netzes

Gruppen von Bewegungsabläufen einzuteilen [AC06] [AC05] [VMIJS12]. Auch in der Literatur zu Modularen Neuronalen Netzen findet man Aussagen, nach denen Modulare Neuronale Netze besser zur Approximation von Funktionen, deren Wertebereich sich abrupt ändert, geeignet sind als herkömmliche monolithische Neuronale Netze [FH96] [JJ93].

Das für diese Aufgabe gewählte Modulare Neuronale Netz besteht aus einem Weichennetz und mehreren Expertennetzen. Dem Weichennetz fällt eine Klassifizierungsaufgabe zu. Als Eingabe bekommt es die aktuelle Position des Agenten (x- und y-Koordinate) und berechnet hieraus als Ausgabe die Nummer des Expertennetzes, welches für die Approximation des Lenkwinkels an dieser Stelle zuständig ist. Dabei wurde beobachtet, dass ein Weichennetz, das neben der x- und der y-Koordinate noch den Fahrzeugwinkel als Eingabe erhält, seine Aufgabe schlechter erfüllt als eines, welches nur auf x- und y-Koordinaten beschränkt ist.

Für das Weichennetz haben sich 5 Neuronen in der verdeckten Schicht als eine geeignete Zahl herausgestellt.

Die Expertennetze sind für die Berechnung des Lenkwinkels zuständig. Ihre Eingaben sind die x-Koordinate, die y-Koordinate und der Fahrzeugwinkel. Aus den Ausgaben wird der Lenkwinkel berechnet. Für die Neuronenanzahl der verdeckten Schicht haben sich 3-5 Neuronen bewährt. Ein Vorteil der Modularen Neuronalen Netze ist es, dass die Anzahl der Neuronen individuell pro Expertennetz eingestellt werden kann, je nachdem wie komplex die in diesem Bereich zu approximierende Funktion ist.

5.5.3 Clustering der Daten

Für das Aufteilen der Daten in Cluster wurden entweder der k-Means Algorithmus oder hierarchisches Clustering verwendet.

Hierarchisches Clustering

Um ein hierarchisches Clustering durchzuführen, stellt Matlab zahlreiche Funktionen zur Verfügung, mit denen die Entfernungen zwischen den einzelnen Datenpunkten und darauf basierend die Entfernungen zwischen und innerhalb eines Clusters bestimmt werden können. Auf diese Weise wird das Dendrogramm aufgebaut, anhand dessen die Cluster im nächsten Schritt gebildet werden.

Hierzu verwendet Matlab den „Inkonsistenz Koeffizienten“. Da die Länge einer Kante von einem Knoten zu seinen Kindern im Dendrogramm anzeigt, wie ähnlich sich die Kinder sind, die zu dem neuen Knoten zusammengefasst werden, kann man nach Sprüngen in der Länge der Kanten suchen, um eine geeignete Aufteilung in Cluster zu bestimmen.

Der Inkonsistenz Koeffizient setzt die Länge der Kanten des aktuellen Knotens in Relation zu der durchschnittlichen Länge der Kanten seiner Kinder. Auf diese Weise kann dann definiert werden, dass immer dort Cluster gebildet werden, wo der Inkonsistenz Koeffizient einen Wert annimmt, der größer als ein bestimmter Cutoff-Wert ist.

Um hierarchisches Clustering für dieses Problem anwenden zu können, musste aufgrund der hohen Anzahl von Trainingsdaten für das Clustern der Daten ein zweistufiger Ansatz verwendet werden, wie er in [VA00] beschrieben wurde. Hierbei wird zunächst eine selbstorganisierende Karte dazu genutzt, um die Trainingsdaten zu komprimieren. Nach dem Training kann jedem zu clusternden Datum ein Neuron der Karten-Schicht zugeordnet werden, dessen Gewichte dem Datum am ähnlichsten sind. Daher kann man die Neuronen der Karten-Schicht als Prototypen ansehen, die jeweils eine bestimmte Menge der originalen Daten repräsentieren. Dabei ist zu beachten, dass es wesentlich mehr Prototypen als spätere Cluster gibt. Im zweiten Schritt werden nun die Prototypen geclustert. Hiervon werden jedoch die Neuronen, denen keine Originaldaten zugeordnet wurden, ausgeschlossen. Im Anschluss werden die originalen Daten dem Cluster ihres jeweiligen Prototyps zugeordnet.

Dieser Ansatz hat den Vorteil, dass er den Rechenaufwand stark reduziert. Da für das Training

der Neuronalen Netze sehr viele Trainingsdaten zur Verfügung standen, konnte durch diese Maßnahme hierarchisches Clustering überhaupt erst in annehmbarer Zeit durchgeführt werden. Weiterhin sind die Prototypen lokale Durchschnittswerte der originalen Daten und daher dazu geeignet, das Clusterverfahren unanfälliger gegen Rauschen zu machen.

K-Means Clustering

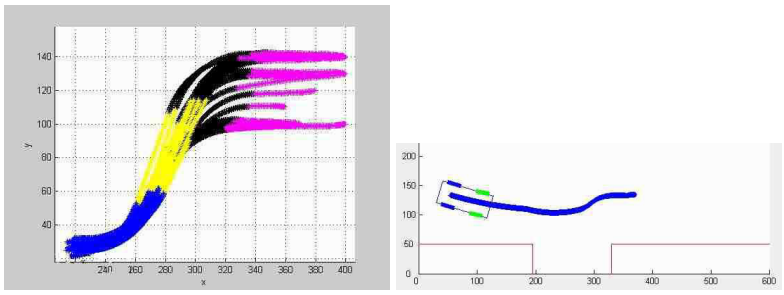
Während beim hierarchischen Clustering die optimale Anzahl der zu bildenden Cluster anhand des Dendrogramms erkannt oder durch den Inkonsisztenz-Koeffizienten gebildet werden kann, muss bei diesem Verfahren die Anzahl der Cluster (k) initial angegeben werden. Eine gängige Strategie, um eine möglichst optimales k zu finden, ist das Ausprobieren von verschiedenen Clusterings mit unterschiedlichem k. Im Anschluss werden diese dann anhand bestimmter Eigenschaften verglichen und das beste Ergebnis ausgewählt [MTCM10] [RT99].

Hierzu wurde der Silhouettenkoeffizient gewählt. Dieser gibt eine Maßzahl für die Qualität des Clusterings an, die unabhängig von der Anzahl der Cluster ist. Für jedes Datum wird die Silhouette bestimmt. Diese nimmt einen Wert aus dem Bereich [-1,1] an und gibt an, wie nahe das Datum den Punkten des benachbarten Clusters ist. Eine 1 bedeutet, dass das Datum weit von den anderen Punkten entfernt ist, während eine -1 aussagt, dass das Datum den Datenpunkten der anderen Cluster sehr nahe – also wahrscheinlich dem falschen Cluster zugeordnet – ist. Der Silhouettenkoeffizient ist das arithmetische Mittel der einzelnen Silhouetten.

5.5.4 Ergebnisse

Es wurden verschiedene modulare Neuronale Netze getestet. Diese unterschieden sich zum einen in der Anzahl der Expertennetze, zum anderen durch die Methode, die für das Clustering der Daten verwendet wurde.

Das beste Ergebnis erzielten modulare Neuronale Netze, bei denen die Lerndaten durch hierarchisches Clustering in 3 Cluster eingeteilt wurden. Diese bestanden folglich aus einem Weichennetz und 3 Expertennetzen. Sie parkten bei bei den automatisierten Tests im Durchschnitt in 98% aller Fälle korrekt in die Parklücke ein. Die Ergebnisse der einzelnen Tests sind im Folgenden aufgeführt.



(a) Ergebnisse des Clusterings (b) Fehlerhafte Trajektorie des Agenten

Abbildung 5.11: Aufteilung der Datenpunkten in 4 Cluster durch das K-Means Clusterverfahren

Ergebnisse bei einer Aufteilung durch K-Means

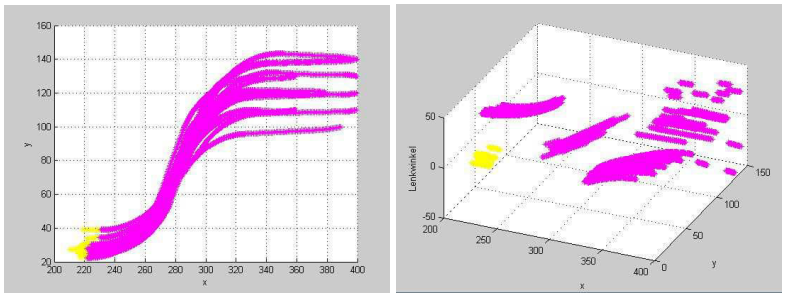
Bei einer Aufteilung der Trainingsdaten durch den K-Means Algorithmus wurden in den automatischen Tests folgende Erfolgsquoten erreicht:

2 Cluster: Der Agent ist in durchschnittlich 48% aller Einparkversuche erfolgreich eingeparkt.

3 Cluster: Der Agent ist in durchschnittlich 88% aller Einparkversuche erfolgreich eingeparkt.

Bei einer Aufteilung in 4 Cluster ist der Agent bei keinem Einparkversuch erfolgreich in die Parklücke navigiert. Die gefahrenen Trajektorien sind in Abbildung 5.11(b) zu sehen. Dieser Fehler lässt sich damit erklären, dass das Weichennetz in einem entscheidenden Punkt das falsche Expertnnetz bestimmt, welches einen nicht passenden Lenkwinkel berechnet. Durch diese Falschberechnung steuert das Fahrzeug für einen kurzen Moment falsch und verlässt hierdurch die trainierte Trajektorie und somit den trainierten Bereich. Da für diesen Bereich jedoch keine Trainingsdaten vorlagen, ist der Verlauf der zu approximierenden Funktion in diesem Bereich unbekannt und die berechneten Werte unzuverlässig. Aus diesem Grund findet das Fahrzeug nicht mehr auf eine Trajektorie zurück, die es in die Parklücke leitet. Die Falschbestimmung des Expertennetzes könnte an der ungünstigen Überschneidung der einzelnen Cluster liegen. Das Ergebnis der Clusterung der Daten in 4 Cluster ist in Abbildung 5.11(a) zu sehen.

Dieses Verhalten lässt sich auch bei einer Aufteilung in noch mehr Cluster beobachten. Das Weichennetz bestimmt für eine kurze Zeit ein falsches Expertennetz, welches daraufhin einen falschen Lenkwinkel berechnet. Hierdurch verlässt der Agent entweder den erlernten Bereich oder er fährt einen Schlenker, den nachfolgende Expertennetze nicht mehr ausgleichen können.



(a) 2 dimensionale Ansicht (b) 3 dimensionale Ansicht

Abbildung 5.12: Aufteilung der Datenpunkten in zwei Cluster durch Hierarchisches Clustering

Ergebnisse bei einer Aufteilung durch Hierarchisches Clustering

Die Ergebnisse des Clusterings in zwei Cluster durch hierarchisches Clustering ist in Abbildung 5.12 zu sehen. Der Algorithmus hat die Daten im wesentlichen in ein großes Cluster und ein äußerst kleines Cluster geteilt, sodass die Ergebnisse der Einparkversuche ähnlich denen mit nur einem Netz sind.

Das Dedrogramm des Clusterings ist in Abbildung 5.13 zu sehen. Dort ist gut zu erkennen, dass bei einer Aufteilung in zwei Cluster eines der Cluster nur aus dem initialen Cluster 12 besteht, also nur aus Trainingsdaten, die einem einzelnen Knoten der vorgelagerten selbstorganisierenden Karte zugeordnet wurden.

Die Ergebnisse des Clusterings in 3 Cluster lieferte die besten Ergebnisse in den automatisierten Tests. Auch bei der Sichtüberprüfung ist ein Agent, dessen modulares Neuronales Netz aus 3 Expertennetzen besteht, die mit diesen in 3 Cluster aufgeteilten Daten trainiert wurden, immer korrekt eingeparkt. Bei den automatisierten Tests wurde im Durchschnitt in 98% aller Fälle korrekt eingeparkt.

Teilt man die Daten weiter auf, nimmt die Erfolgsrate wieder ab. So beträgt sie bei einer Aufteilung in 4 Cluster nur noch 65%. Bei einer Aufteilung in noch mehr Cluster entstehen Fehler häufig dadurch, dass das Weichennetz an einem entscheidenden Punkt des Einparkvorgangs kurzzeitig das falsche Expertennetz bestimmt. Da dem falsch bestimmten Expertennetz für diesen Bereich keine Trainingsdaten vorlagen, sind seine Ausgaben nicht zuverlässig und das Fahrzeug navigiert aus der Trajektorie. Dieser Fehler kann in den meisten Fällen durch das Expertennetz, welches kurz danach richtig bestimmt wird, nicht mehr ausgeglichen werden.

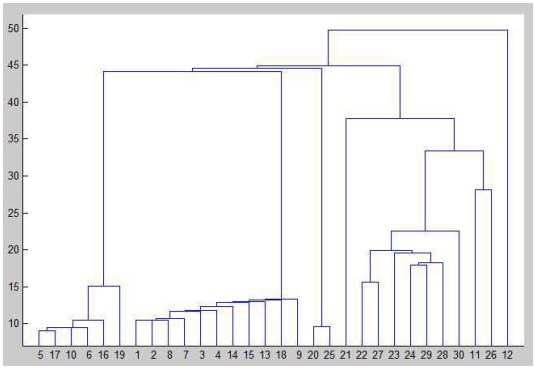


Abbildung 5.13: Dendrogramm der geclusterten Daten

Automatische Bestimmung der optimalen Clusteranzahl

Versuche, die optimale Aufteilung der Trainingsdaten vollkommen automatisch zu gestalten, waren nicht erfolgreich. Für das Aufteilen der Daten durch den K-Means Algorithmus wurde der Silhouettenkoeffizient zur Bestimmung der optimalen Clusteranzahl verwendet. Dieser war maximal bei einem Wert von k = 4. Einparkversuche haben jedoch gezeigt, dass bei einer Clusterung durch den K-Means Algorithmus für diese Aufgabe 3 Cluster zu bevorzugen sind, während 4 Cluster immer zu einem Scheitern des Einparkvorgangs geführt haben.

Beim Hierarchischen Clustering gibt das Dendrogramm Hinweise darauf, wie man die Trainingsdaten Clustern könnte. Das Dendrogramm der mit Hilfe einer selbstorganisierenden Karte vorverarbeiteten Lerndaten ist in Abbildung 5.13 zu sehen. Aus diesem Dendrogramm ist eine Einteilung in 3 Cluster, die sich als für diese Aufgabe geeignet erwiesen hat, weniger heraus zu sehen.

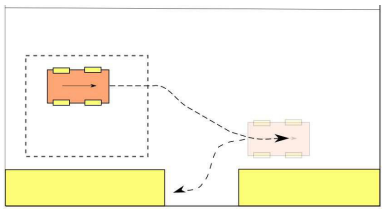


Abbildung 5.14: Vorwärts einparken

5.6 Szenario 5

5.6.1 Zielstellung und Randbedingungen

Bei diesem Szenario sollte der Agent folgende Aufstellung erlernen:

Zielstellung:

- Der Agent befindet sich vor der Parklücke.
- Um in die Parklücke zu gelangen, muss zunächst hinter diese navigiert und von dort aus rückwärts eingeparkt werden (siehe Abbildung 5.14).

Randbedingungen:

- Die Startposition des Agenten kann in x- und y- Richtung variieren. Ebenso der Winkel, in dem sich das Fahrzeug anfangs befindet.
- Der Agent soll ohne Rangieren in die Parklücke einparken.
- Zu einem Zustand sollen eine Richtung und der Lenkwinkel als Ausgabe erlernt werden.
- Der Zustand wird durch die Fahrzeugposition (x-Koordinate, y-Koordinate), den Fahrzeugwinkel ϕ und die aktuelle Einparkphase s beschrieben.

Die Startposition des Agenten ist in Abbildung 5.14 zu sehen.

5.6.2 Funktionsapproximation

Auch bei diesem Versuch wurden zur Funktionsapproximation modulare Neuronale Netze verwendet. Der Aufbau der Netzes ist in Abbildung 5.15 zu sehen.

Dort ist ebenfalls zu sehen, dass es sich im Prinzip um den gleichen Aufbau wie bei Szenario 4 handelt. Es wurde lediglich eine weitere Ebene eingefügt.

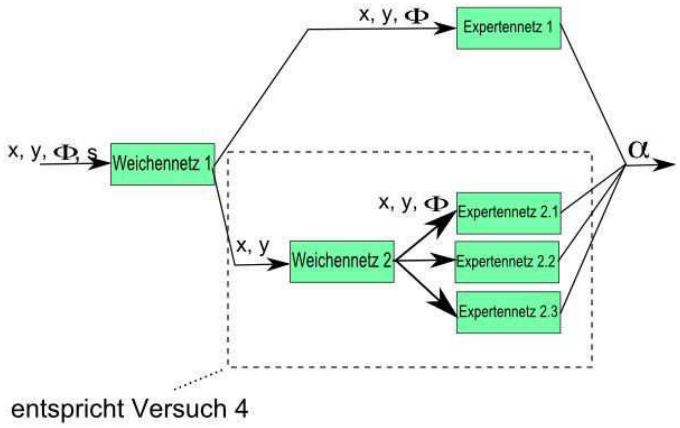


Abbildung 5.15: Aufbau des modularen Neuronalen Netzes

Der Einparkvorgang wurde zunächst in zwei Teile geteilt:

1. Fahrt hinter die Parklücke.
2. Navigation in die Parklücke.

Das Problem der Navigation in die Parklücke, wenn sich das Fahrzeug bereits hinter der Parklücke befindet, wurde in Szenario 4 behandelt. Dieser Aufbau wurde nun um ein weiteres Weichennetz (Weichennetz 1) erweitert. Dieses Netz hat die Aufgabe, herauszufinden, ob das Fahrzeug weiterhin nach vorne zur Parklücke navigieren, oder ob es bereits rückwärts in die Parklücke hinein manövrieren muss. Hierzu bekommt es 4 Eingaben:

1. x-Koordinate
2. y-Koordinate
3. Lenkwinkel ϕ
4. Einparkphase s

Dies entspricht auch in etwa dem Aufbau von Weichennetz 2. Allerdings wird bei diesem ersten Weichennetz noch zusätzlich benötigt, in welcher Einparkphase das Fahrzeug sich aktuell befindet. Einparkphase 1 entspricht hierbei der Vorwärtsfahrt bis hinter die Parklücke. Einparkphase 2 ist die Phase, in der das Fahrzeug in die Lücke hinein manövriert. Die aktuelle

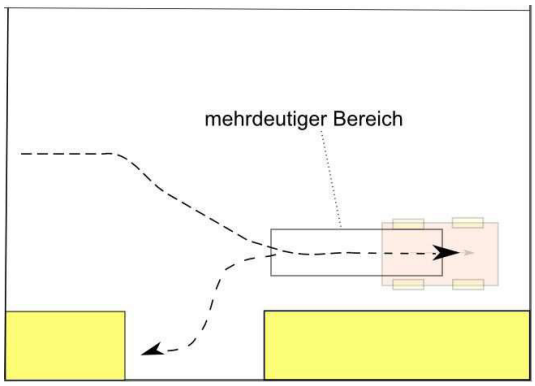


Abbildung 5.16: Mehrdeutiger Bereich

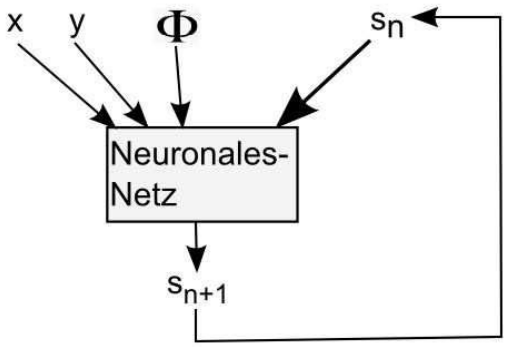


Abbildung 5.17: Modell von Weichennetz 1

Einparkphase ist hierbei eine unverzichtbare Eingabe, da hierdurch Mehrdeutigkeiten, in welche Richtung sich das Fahrzeug bewegen muss, verhindert werden. (siehe Abbildung 5.16). Der Fahrzeugwinkel wurde als weitere Eingabe hinzugefügt, da sich herausgestellt hat, dass sonst in manchen Fällen zu schnell von der Vorwärtsfahrt in den eigentlichen Einparkvorgang übergegangen wird. Die Ursache hierfür liegt darin, dass ohne Beachtung des Fahrzeugwinkels nur auf die Fahrzeugpositon, nicht jedoch darauf, ob sich das Fahrzeug annähernd parallel zur Parklücke befindet, oder ob noch ein weiter korrigiert werden muss, geachtet wird. Das Weichennetz berechnet aus seinen Ausgaben in welchem Zustand sich der Agent im nächsten Schritt befindet (siehe Abbildung 5.17). Verbunden mit der Einparkphase ist die Richtung, in die sich der Agent bewegt:

Einparkphase 1: vorwärts

Einparkphase 2: rückwärts

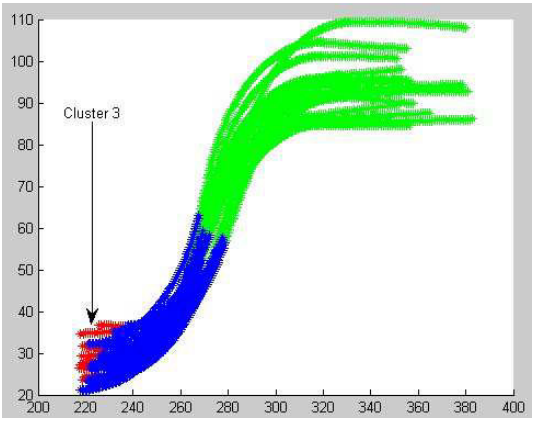


Abbildung 5.18: Clustering der Daten für ein Subnetz

Außerdem bestimmt die aktuelle Einparkphase, welches Subnetz im Moment für die Berechnung des Lenkwinkels zuständig ist.

Das Subnetz, welches für die Navigation in die Parklücke zuständig ist, entspricht im Aufbau dem modularen Neuronalen Netz aus Szenario 4. Für die Annäherung des Lenkwinkels in Einparkphase 1 (Vorwärtsfahrt) wurde anstatt eines modularen Neuronalen Netzes wieder ein einzelnes vorwärtsgerichtetes Neuronales Netz (Expertennetz 1) genutzt, da die in diesem Bereich zu approximierende Funktion eher glatt ist.

5.6.3 Ergebnisse

Das beste Ergebnis bei diesem Versuch war eine Erfolgsrate von 73%. Für das Subnetz, welches für die Navigation in die Parklücke zuständig war, wurden zunächst drei Expertennetze genutzt, da sich diese Aufteilung in Szenario 4 als am besten geeignet herausgestellt hat. Die dazugehörigen Cluster sind in Abbildung 5.18 zu sehen. Allerdings fällt bei dieser Aufteilung auf, dass Cluster 3 bei diesen Trainingsdaten aus sehr wenigen Datensätzen besteht. Dies ist ebenfalls in Abbildung 5.18 zu sehen. Hat der Agent bereits eine gute Einparkposition erreicht, bevor er in den Bereich von Expertennetz 2.3 kommt, welches mit den Daten aus Cluster 3 trainiert wurde, parkt der Agent häufig erfolgreich ein. Ist der Einparkvorgang in diesem Bereich noch nicht beendet, berechnet Expertennetz 3 Lenkwinkel, die zu einem Schlenker am Ende der Trajektorie führen, welcher wiederum in einer Kollision mit den Parklückenbegrenzungen resultiert. Die Erfolgsrate eines solchen Netzes, welches 3 Expertennetze zur Navigation in die Parklücke benutzt, lag im Durchschnitt bei 42%.

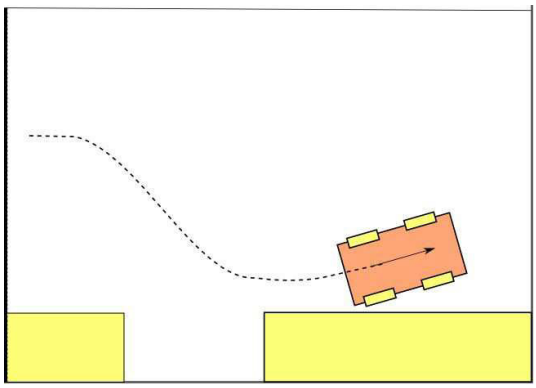


Abbildung 5.19: Häufig beobachteter Fehler

Bei Versuchen, bei denen für diese Aufgabe nur 2 Expertennetze genutzt wurden, lag die Durchschnittliche Erfolgsrate bei 71%. Die Daten aus Cluster 3 wurden dabei verworfen und nicht für das Training benutzt.

Ein Fehler, der dennoch häufig auftrat, war, dass sich der Agent während der Vorwärtsfahrt nicht rechtzeitig parallel zur Parklücke angeordnet hat. Der Agent ist weiter nach vorne gefahren und hat so den Fahrzeugwinkel korrigiert. In manchen Fällen wurde dabei zu lange korrigiert, bevor der Zustand in Einparkphase 2 gewechselt ist, sodass der Agent erneut nicht parallel zur Parklücke stand (siehe Abbildung 5.19). Aus dieser Position ist der Agent häufig während des Navigierens in die Parklücke mit einem Hindernis kollidiert.

6 Zusammenfassung

Im Rahmen dieser Arbeit wurden verschiedene Verfahren untersucht, um ein autonomes Fahrzeug durch überwachtes Lernen das Einparken in eine Parklücke „beizubringen“. Um dies zu ermöglichen, musste eine Funktion angenähert werden. Hierdurch konnte zu dem aktuellen Zustand (der Position des Fahrzeugs) eine konkrete Aktion (Lenkwinkel und in manchen Fällen eine Richtung) berechnet werden.

Zur Darstellung des aktuellen Zustands wurden zunächst von Sensoren gemessene Entfernungen genutzt.

Die zweite Möglichkeit war die Darstellung des Zustands durch die Fahrzeugposition und den Winkel (die Schrägstellung) des Fahrzeugs zu beschreiben. Diese Möglichkeit zeichnet sich dadurch aus, dass sie einen Zustand im Gegensatz zur Beschreibung mit den Sensordaten immer eindeutig und sehr präzise beschreibt.

Zur Funktionsapproximation wurden Neuronale Netze genutzt. Hierbei haben sich modulare Neuronale Netze den herkömmlichen monolithischen Netzen als überlegen erwiesen. Dies kann damit begründet werden, dass bei einem Einparkvorgang auf engem Raum und teilweise sogar dann, wenn sich das Fahrzeug nicht bewegt, gelenkt wird. Aus diesem Grund kann es vorkommen, dass sich der zu berechnende Lenkwinkel zwischen zwei benachbarten Zuständen sprunghaft ändert. Funktionen mit einem solchen Verhalten lassen sich in der Regel besser mit modularen Neuronalen Netzen approximieren.

Das beste Ergebnis wurde dabei in dem Szenario erzielt, bei dem sich das Fahrzeug den Einparkvorgang hinter der Parklücke begonnen hat und rückwärts in einem Zug in diese einparken sollte. Zur Funktionsapproximation wurde ein modulares Neuronales Netz mit einem Weichennetz und 3 Expertennetzen verwendet, wobei die Trainingsdaten zunächst durch hierarchisches Clustering aufgeteilt wurden. Das Fahrzeug ist in 98% aller automatischen Einparkversuche korrekt eingeparkt.

Bei einer komplexeren Aufgabenstellung, bei der das Fahrzeug sich zunächst vor der Parklücke befindet und zunächst hinter diese gelangen muss, um mit dem Einparkvorgang zu beginnen, wurden hingegen wieder schlechtere Ergebnisse erzielt. Das Fahrzeug ist in 71% aller automatischen Einparkversuche korrekt eingeparkt.

7 Ausblick

7.1 Kombination mit Regelungstechnik

Eine Möglichkeit, die bisherigen Verfahren zu erweitern und dadurch eventuell zu optimieren, wäre die Kombination der bisherigen Systeme mit Regelungstechnik.

Eine Schwäche der bisherigen Verfahren ist es, dass das System zur Berechnung der Steuergrößen, sobald das Fahrzeug durch einen Fehler von der erlernten Trajektorie abkommt – sich also in einem Bereich befindet, für den keine Trainingsdaten vorliegen – in diesen Bereichen willkürliche Ausgaben produzieren kann. Es reicht folglich ein kleiner Fehler, durch welchen das Fahrzeug von der Soll-Trajektorie abweicht, um den gesamten folgenden Einparkvorgang negativ zu beeinflussen. Sobald das Fahrzeug einmal aus dem gelernten Bereich gefahren ist, besteht kaum noch eine Chance, dass es wieder auf eine der Soll-Trajektorien zurückfindet.

Eine Möglichkeit, um mit diesem Verhalten besser umzugehen, wäre es, zu erkennen, wenn das Fahrzeug von den möglichen Soll-Trajektorien abweicht und in einem solchen Fall den Einparkvorgang zu beenden, bzw. neu zu starten.

Eine weitere Möglichkeit könnte die Kombination der bisherigen Systeme mit Regelungstechnik sein. Zusätzlich zum System, welches für die Berechnung des Lenkwinkels zuständig ist, könnte ein weiterer Controller genutzt werden. Dieser müsste Abweichungen des aktuellen Zustands von den gelernten Trajektorien festzustellen und durch eine geeignete Regelung dazu beitragen, das Fahrzeug zurück auf die Soll-Trajektorie zu manövrieren.

7.2 Überlappende Cluster

Wenn zur Funktionsapproximation modulare Neuronale Netze verwendet werden, treten Fehler häufig beim Übergang von einem Expertennetz zum nächsten Expertennetz auf. Die Ursachen hierfür sind, dass durch das Aufteilen der Trainingsdaten in Cluster zusätzliche Randbereiche entstehen. Wenn das Weichennetz nicht im genau richtigen Moment zwischen benachbarten

Netzen umschaltet, kann es sein, dass das aktuelle Expertennetz über diesen trainierten Bereich hinaus die Funktion approximieren muss. Dies geschieht dann, wenn das Weichennetz zu früh umschaltet. In diesem Fall befindet sich das Fahrzeug noch im Bereich von Expertennetz n, das Weichennetz hat jedoch bereits Expertennetz n+1 für die Funktionsapproximation bestimmt, welchem für diesen Bereich jedoch noch keine Lerndaten vorlagen.

Die andere Möglichkeit ist, dass das Weichennetz zu spät umschaltet. In diesem Fall wird noch Expertennetz n verwendet, obwohl sich das Fahrzeug bereits im Bereich von Expertennetz n+1 befindet. Auch in diesem Fall befindet sich das Fahrzeug also in einer Position, für die dem aktuell bestimmten Expertennetz keine Lerndaten vorlagen.

Eine Möglichkeit, um mit diesem Problem umzugehen, könnte sein, die einzelnen Cluster überlappend zu gestalten, sodass die Punkte in den Grenzbereichen zu zwei benachbarten Clustern gehören und folglich auch von zwei Expertennetzen erlernt werden, sodass das Umschalten zwischen Expertennetzen durch das Weichennetz nicht mehr auf den Punkt genau erfolgen muss, sondern kleine Abweichungen von den Experten toleriert werden.

Literaturverzeichnis

[AC05] J. Aleotti and S. Caselli. Trajectory clustering and stochastic approximation for robot programming by demonstration. In *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, pages 1029–1034, 2005.

[AC06] Jacopo Aleotti and Stefano Caselli. Robust trajectory learning and approximation for robot programming by demonstration. *Robotics and Autonomous Systems*, 54(5):409–413, 2006.

[Alp08] E. Alpaydın. *Maschinelles Lernen.* Oldenbourg, München, 2008.

[Alp10] T. Alpers. Modellierung eines Einparkassistenten für ein autonomes Fahrzeug implementiert auf einer SoC-Plattform. Bachelor's thesis, HAW Hamburg, Hamburg, 2010.

[Aza00] F. Azam. *Biologically Inspired Modular Neural Networks.* PhD thesis, Faculty of the Virginia Polytechnic Institute and State University, Blacksburg, 2000.

[Ben95] Younès Bennani. A modular and hybrid connectionist system for speaker identification. *Neural Computation*, 7(4):791–798, 1995.

[Ber98] M. Bernreuther. Funktionsapproximation mit Hilfe von künstlichen neuronalen Feedforward-Netzen. http://elib.uni-stuttgart.de/opus/volltexte/2000/668/pdf/FBI98.pdf, 1998. abgerufen am: 11.01.2013.

[Bit02] O Bittel. NeuroNetze4. http://www-home.fh-konstanz.de/~bittel/nnfl/NeuroNetze_4.pdf, 2002. abgerufen am: 30.05.2013.

[Cal03] R. Callan. *Neuronale Netze im Klartext.* Pearson Studium, München, 2003.

[e.V13] Deutscher Verkehrssicherheitsrat e.V. Bester Beifahrer: Der Leitfaden zu Fahrerassistenzsystemen. http://www.bester-beifahrer.de/

fileadmin/redaktion/Downloads/FAS-verstehen.pdf, 2013. abgerufen am: 27.02.2013.

[FH96] Meng-Hock Fun and M.T. Hagan. Modular neural networks for friction modeling and compensation. In *Control Applications, 1996., Proceedings of the 1996 IEEE International Conference on*, pages 814–819, 1996.

[HOHK06] M.R. Heinen, F.S. Osorio, F.J. Heinen, and C. Kelber. Seva3d: Using arti cial neural networks to autonomous vehicle parking control. In *Neural Networks, 2006. IJCNN '06. International Joint Conference on*, pages 4704–4711, 2006.

[JJ93] R.A. Jacobs and M.I. Jordan. Learning piecewise control strategies in a modular neural network architecture. *Systems, Man and Cybernetics, IEEE Transactions on*, 23(2):337–345, 1993.

[LC12] U. Lämmel and J. Cleve. *Künstliche Intelligenz*. Hanser, München, 2012.

[Mat10a] Mathworks. Matlabhilfe: Neural Network Toolbox, 2010.

[Mat10b] Mathworks. Matlabhilfe: Statistics Toolbox, Cluster Analysis, 2010.

[Mei08] A. Meisel. NeuronaleNetze_ Vortrag. http://www.informatik.haw-hamburg.de/fileadmin/Homepages/ProfMeisel/Vortraege/NeuronaleNetze_Vortrag.pdf, 2008. abgerufen am: 06.03.2013.

[Mer08] D. Merkl. Clustering. http://www.ec.tuwien.ac.at/~dieter/teaching/dm08-clustering.pdf, 2008. abgerufen am: 13.05.2013.

[MTCM10] M. Ming-Tso Chiang and B. Mirkin. Intelligent choice of the number of clusters in k-means clustering: An experimental study with different cluster spreads. *Journal of Classification*, 27:3–40, 2010.

[NYWI08] H. Nakamura, Y. Yafuso, K. Watanabe, and H. Igarashi. A study on automatic parking for automobiles using rational policy making method. In *Soft Computing in Industrial Applications, 2008. SMCia '08. IEEE Conference on*, pages 7–12, 2008.

[OHF02] F.S. Osorio, F. Heinen, and L. Fortes. Autonomous vehicle parking using finite state automata learned by j-cc artificial neural nets. In *Neural Networks, 2002. SBRN 2002. Proceedings. VII Brazilian Symposium on*, pages 196–, 2002.

[Rie12] L. Rieken. Einparken lernen durch Simulation. Bachelor's thesis (Studienarbeit), HAW Hamburg, Hamburg, 2012.

[RT99] S. Ray and R. Turi. Determination of number of clusters in k-means clustering and application in colour image segmentation. 1999.

[SGD12a] N.N. Samani, J. Ghaisari, and M. Danesh. Autonomous parallel parking of a vehicle in a limited space using a rbf network and a feedback linearization controller. In *Computer and Knowledge Engineering (ICCKE), 2012 2nd International eConference on*, pages 117–122, 2012.

[SGD12b] N.N. Samani, J. Ghaisari, and M. Danesh. Autonomous parallel parking of a vehicle in a limited space using a rbf network and a feedback linearization controller. In *Computer and Knowledge Engineering (ICCKE), 2012 2nd International eConference on*, pages 117–122, 2012.

[VA00] J. Vesanto and E. Alhoniemi. Clustering of the self-organizing map. *Neural Networks, IEEE Transactions on*, 11(3):586–600, 2000.

[VMIJS12] A. Vakanski, I. Mantegh, A. Irish, and F. Janabi-Sharifi. Trajectory learning for robot programming by demonstration using hidden markov model and dynamic time warping. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 42(4):1039–1052, 2012.

[Wag99] H. Wagner. *Modularisierung von Neuronalen Netzen.* PhD thesis, Universität Bielefeld, Bielefeld, 1999.

[WHW12] H. Winner, S. Hakuli, and G. Wolf. *Einparkassistenz*, pages 471–477,. Vieweg + Teubner, Wiesbaden, 2012.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 1. Oktober 2013 Sebastian Krome