Technology
Arts Sciences
TH Köln

Masterarbeit

Untersuchung des Potentials Neuronaler Netze für Regelungsprozesse am Beispiel eines Betonverteilers

Vorgelegt an der TH Köln
im Studiengang
Informatik/Computer Science

ausgearbeitet von
Joram Kay Markert

Erstprüfer: Prof. Dr. Heide Faeskorn-Woyke
Zweitprüfer: Dipl. Inf. Stefan Blauel

Technische Hochschule Köln
Fakultät für Informatik und
Ingenieurwissenschaften

In Kooperation mit
Unitechnik Systems GmbH

Gummersbach, im April 2018

Zusammenfassung In dieser Arbeit wird untersucht, ob sich Neuronale Netze in Regelungsprozesse integrieren lassen, um diese zu optimieren. Das dafür erarbeitete Konzept wird anhand der Regelung eines Betonverteilers für die Produktion von Betonfertigteilen erläutert.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Das Thema Künstliche Intelligenz (KI) gewann in den vergangenen Jahren an Relevanz. So berichten beispielsweise Online-Nachrichtenportale immer häufiger über Entwicklungen auf diesem Gebiet.[1] Häufig ist mit *Künstlicher Intelligenz* das gemeint, was auch E. Rich darunter versteht:

> *Artificial intelligence is the study of how to make computers do things which, at the moment, people do better.* [39]

Trotz der großen Rechenleistung moderner Computer gibt es viele Aufgaben, bei denen Menschen und Tiere den Computern überlegen sind. Eine Stärke menschlicher Intelligenz ist die Anpassungs- und Lernfähigkeit, welche ein Computer nicht in diesem Maß besitzt [9]. Techniken der Künstlichen Intelligenz sollen Computern deswegen in der Regel dazu verhelfen, Kompetenzen der natürlichen Intelligenz zu erwerben.

Zahlreiche Anwendungen und Produkte wurden durch Künstliche Intelligenz erst möglich. Dazu gehören z.B. Fortschritte in dem Bereich der Spracherkennung, welche eine Voraussetzung für digitale Assistenten wie Siri (Apple), Alexa (Amazon) und Cortana (Microsoft) waren. Techniken der Künstlichen Intelligenz bleiben somit nicht nur theoretisch interessant, sondern es kommen auch immer mehr Menschen damit in Berührung [44].
Neben der Spracherkennung gibt es auch Anwendungsfälle im Bereich der Bilderkennung und -verarbeitung. Seit 2012 wird beispielsweise der Bilderkennungswettbewerb *Large Scale Visual Recognition Challenge (ILSVRC)*[2] von *deep learning-*

[1] Ergab eine Analyse von Artikeln der Kategorie „Künstliche Intelligenz“ verschiedener Online-Zeitschriften. Siehe Abschnitt A.1

[2] siehe `http://www.image-net.org/challenges/LSVRC/` [abgerufen: 04.04.18]

Algorithmen dominiert, welche in der Lage sind, große künstliche Neuronale Netze mit mehr als drei Schichten und hunderttausenden Neuronen zu trainieren [57] [53, S. 8]. Auch Adobes *Project Scribbler*[3] aus dem Jahr 2017 welches Schwarz-Weiß-Bilder einfärben kann, verwendet dafür Neuronale Netze.

Trotz dieser Entwicklungen gibt es gerade bei komplexen Vorgängen Defizite in Bezug auf die Autonomie und Lernfähigkeit von Produktionsanlagen [22, S. 43]. Eine Studie aus dem Jahr 2017 zeigt außerdem, dass Unternehmen das größte Potential von Künstlicher Intelligenz in dem Bereich der Automatisierungstechnologien sehen [43, S. 19]. Das Defizit und ein potentieller Lösungsansatz sind somit möglicherweise bekannt.

1.1 Motivation und Zielsetzung

Die erwähnte Studie ermittelte außerdem, dass bei 40% der Befragten, die in ihrem Unternehmen keine Künstliche Intelligenz verwenden, den Einsatz von KI geplant ist [43, S. 14]. Ähnliche Zahlen nennt auch eine Gartner-Studie, wo insgesamt 46% die Verwendung von KI planen [10].
Bereits 1990 stellte Andrew G. Barto mehrere Ansätze vor, wie Neuronale Netze in eine Regelung integriert werden können [5]. Es folgten weitere Veröffentlichungen wie die von Neumerkel und Lohnert [35]. Einer der Hauptgründe, warum die Technik trotz des erkannten Potentials nur in 4% der befragten Unternehmen zum Einsatz kommt, sind laut Gartner fehlende Kompetenzen [10, 12]. Sopra Steria ermittelte, dass die Kompetenzen vor allem bei den Berufseinsteigern und erfahrenen Fachkräften liegen, der Einsatz von KI aber hauptsächlich durch die Geschäftsführung vorangetrieben wird.

Das Ziel dieser Arbeit ist es deswegen, insbesondere das Potential künstlicher Neuronaler Netze (KNN) für die Optimierung von Regelungsprozessen zu prüfen. Außerdem soll eine Herangehensweise erarbeitet werden, wie Neuronale Netze in Regelprozesse integriert werden können, um deren Potential nutzbar zu machen.

[3] siehe `https://research.adobe.com/project/scribbler-controlling-deep-image-synthesis-with-sketch-and-color/` [abgerufen: 04.04.18]

1.2 Anwendungsfall Betonverteiler

Als konkreter Anwendungsfall für diese Untersuchung dient die Produktion von Betonfertigteilen. Dabei wird auf Stahlpaletten, welche mit Schalung und Bewehrung[4] bestückt sind, Beton mithilfe eines automatischen Betonverteilers ausgetragen. Die Aufgabe der Regelung eines solchen Verteilers ist es, den Beton möglichst genau den Vorgaben entsprechend auszubringen. Zudem sollte dies in möglichst kurzer Zeit erfolgen. Herausfordernd ist vor allem, auf die schwankende Konsistenz des Betons zu reagieren und Messstörungen zu filtern, damit der Austrag möglichst genau mit den Vorgabewerten übereinstimmt.
In dieser Arbeit wird die Optimierbarkeit dieser Regelung mithilfe Neuronaler Netze untersucht. Das dafür gewählte Vorgehen und die Inhalte dieser Ausarbeitung werden zunächst in dem folgenden Abschnitt beschrieben.

1.3 Aufbau der Arbeit

Bevor in Kapitel 3 eine Vorgehensweise vorgestellt wird, wie sich ein Neuronales Netz in eine Regelung integrieren lässt, geht es in dem nun folgenden Kapitel um die Grundlagen dafür.
Zunächst ist in Kapitel 2 erläutert, wie der Betonverteiler und dessen Regelung aktuell aufgebaut sind. Anschließend wird auf einige Schwerpunkte im Bereich künstlicher Neuronaler Netze eingegangen, bevor es am Ende des Grundlagenteils um die Kombination von Regelung und Neuronalen Netzen geht.
Nachdem der Ansatz für die Erstellung einer neuronalen Regelung in Kapitel 3 beschrieben wurde, erläutert Kapitel 4 die Umsetzung des Ansatzes in Bezug auf den Anwendungsfall des Betonverteilers.
Die Arbeit schließt mit den erreichten Ergebnissen (Kapitel 5) sowie einem Ausblick (Kapitel 6) auf mögliche Weiterführungen der Untersuchungen.

[4] „Eine Bewehrung [...] kann aus Matten, Stäben oder Geflechten aus Stahl bestehen. Die Bewehrung wird in den Beton eingelegt und erhöht [...] die Belastbarkeit der Bauteile.“ [2]

Kapitel 2

Grundlagen

Um später in der Arbeit die Möglichkeiten einer neuronalen Regelung für den Betonverteiler zu untersuchen, ist es notwendig, zunächst die Grundlagen dafür zu betrachten. In diesem Kapitel geht es neben der aktuellen Regelung des Betonverteilers um die Forschung im Bereich künstlicher Neuronaler Netze und deren Einsatzmöglichkeiten in Regelprozessen.

2.1 Herstellung von Betonfertigteilen

Das Austragen des Betons durch den Verteiler ist nur ein Teil des Herstellungsprozesses von Betonfertigteilen. Vor der Erläuterung des Betonverteilers und des Betoniervorgangs, wird zunächst ein Überblick über die Produktionsanlage gegeben.

2.1.1 Überblick über den Produktionsprozess

Zu Beginn des Produktionsprozesses werden die Betonteile mit einer CAD-Software als 3D Modelle am Computer erstellt. Diese Modelle bilden die Grundlage für alle weiteren Schritte.
Der *Prozessleitrechner* verarbeitet diese Daten und sendet entsprechende Steuerbefehle an das *Palettenumlaufsystem*, den *Schalungsroboter*, die *Bewehrungsmaschine* sowie den *Betonverteiler*.
Die Betonteile werden auf Stahlpaletten gegossen. Das Palettenumlaufsystem transportiert diese Stahlpaletten in der Fertigungshalle beispielsweise zum Betonierplatz, einer Bearbeitungsstation oder in die Trockenkammer. Soll auf einer Palette Beton ausgetragen werden, befestigt der Schalungsroboter als nächstes die Schaleisen mit

Magneten auf der Palette. Es kann auch vorkommen, dass Löcher für Hohlwanddosen oder ähnliche Elemente ausgespart bleiben. Dazu wird deren Position von einem Laser auf der Palette angezeigt und manuell beispielsweise ein Stück Styropor platziert. Das Aufbringen der Bewehrung erfolgt per Hand, teilweise aber auch durch die Bewehrungsmaschine. [15, S. 5-9] In Abbildung 2.1 ist eine solche Palette mit Schalung und Bewehrung dargestellt.

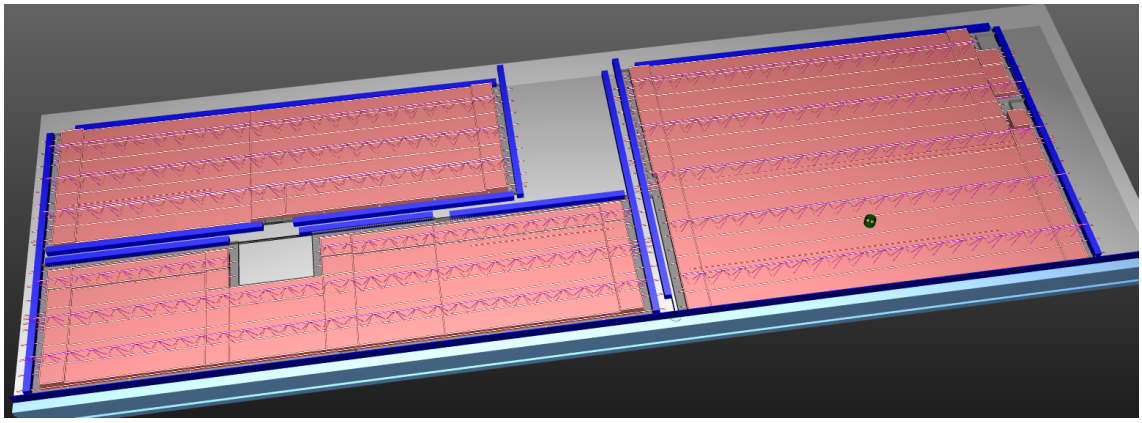

Abbildung 2.1: Ansicht einer Palette im Leitrechner mit drei Betonelementen, Betoniersätzen (rosa), Schalung (blau) und Bewehrung (lila). Auf der rechten Palette ist eine Aussparung für eine Hohlwanddose vorgesehen.

Nachdem Schalung und Bewehrung platziert sind, trägt der Betonverteiler den Beton aus. In der obigen Abbildung (2.1) sind die Betoniersätze rosa dargestellt. Diese geben dem Verteiler vor, an welcher Position wie viel Beton platziert werden soll. In Abschnitt 2.1.3 wird später noch detailliert auf diesen Prozess eingegangen.
Die Betonmasse soll letztendlich gleichmäßig über das Element verteilt sein. Nach dem Austragen des Betons gibt es allerdings noch Unebenheiten, weswegen die gesamte Palette in Schwingung versetzt wird. Der Produktionsvorgang ist beendet, wenn das Element in einer Trockenkammer ausgehärtet ist. [15, S. 10-12]

2.1.2 Aufbau des Betonverteilers

In diesem und dem nächsten Abschnitt geht es ausschließlich um den Betoniervorgang. Die restlichen Produktionsschritte werden nicht weiter erläutert, da sich diese Arbeit nur auf den Regelungsprozess des Betonverteilers konzentriert.

Der Betonverteiler besteht aus einem trichterförmigen Kübel (siehe Abbildung 2.2), in welchen der fertig gemischte Beton eingefüllt wird. Der Kübel liegt auf drei Ge-

wichtsmesszellen auf, worüber sich die Masse des ausgetragenen Betons zwischen mehreren Messungen bestimmen lässt. Am unteren Ende des Trichters befinden sich mehrere Auslassöffnungen, die mit je einer Klappe geschlossen oder geöffnet werden können. Bevor der Beton allerdings durch die Öffnungen nach außen gelangt, passiert er abhängig von dem Modell des Verteilers eine Stachelwalze oder eine Förderschnecke. Durch die Regelung der Walzen- bzw. Schneckendrehzahl lässt sich somit die Austragsmenge beeinflussen. Eine Stachelwalze nimmt die komplette Breite des Trichters ein, wohingegen es bei der Variante mit den Förderschnecken eine Schnecke für jede Öffnung gibt.

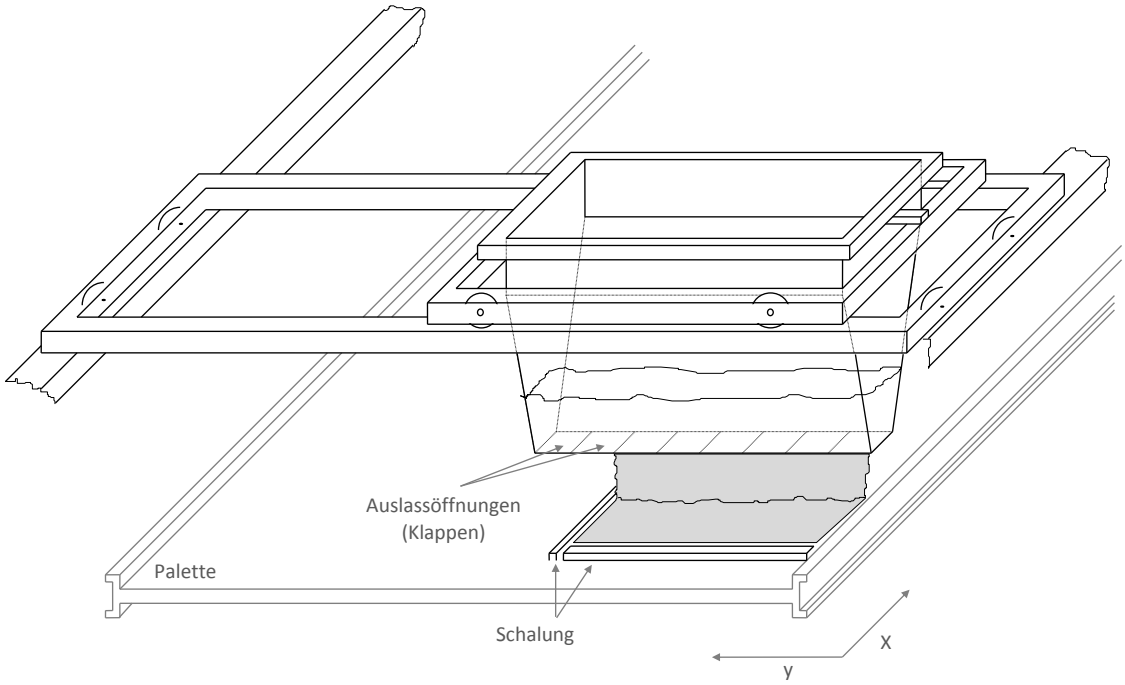

Abbildung 2.2: Schematischer Aufbau eines Betonverteilers nach HEIENBROK [15]

Wie in der obigen Abbildung zu sehen ist, kann der Verteiler in Längsrichtung der Palette oder quer zur Palette bewegt werden.

2.1.3 Regelungsprozess des Betonverteilers

In *Entwicklung und Inbetriebnahme einer Mehrgrößendosierregelung für ein Betonfertigteilwerk* stellte K. HEIENBROK einen optimierten Regelkreis vor, welcher aktuell in abgewandelter Form eingesetzt wird [15, S. 59]. Die folgende Abbildung (2.3) zeigt die Übersicht eines aktuellen Regelkreises.

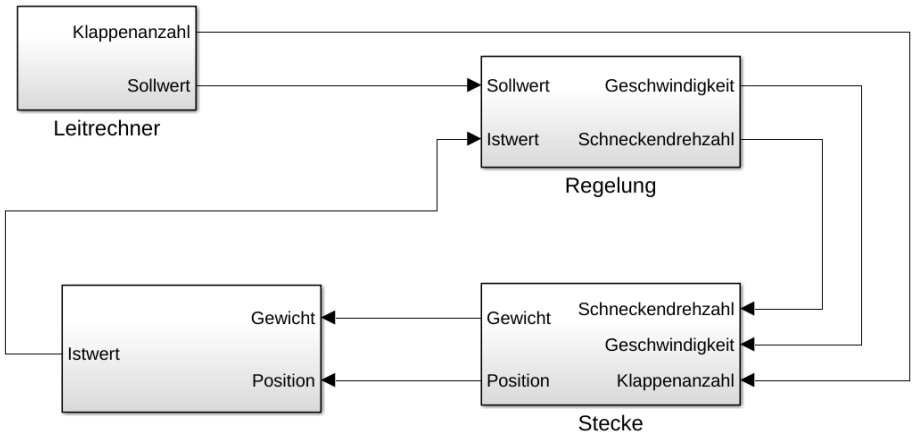

Abbildung 2.3: Ein vereinfachtes Blockschaltbild des konventionellen Regelkreises des Betonverteilers.

Zu Beginn eines Betoniervorgans werden von dem Leitrechner *Betoniersätze* vorgegeben. Diese enthalten die folgenden Informationen [15, S. 17]:

- Start- und Zielposition (X- und Y-Achse)
- Art des Datensatzes (Betonieren oder Positionieren)
- Status der Klappen (offen/geschlossen)
- Austragsverhältnis (Verhältnis zwischen den Drehzahlen der einzelnen Schnecken)
- auszutragendes Volumen

In Abbildung 2.4 ist die grafische Ansicht der Betoniersätze zu sehen, wie sie im Leitrechner angezeigt wird.

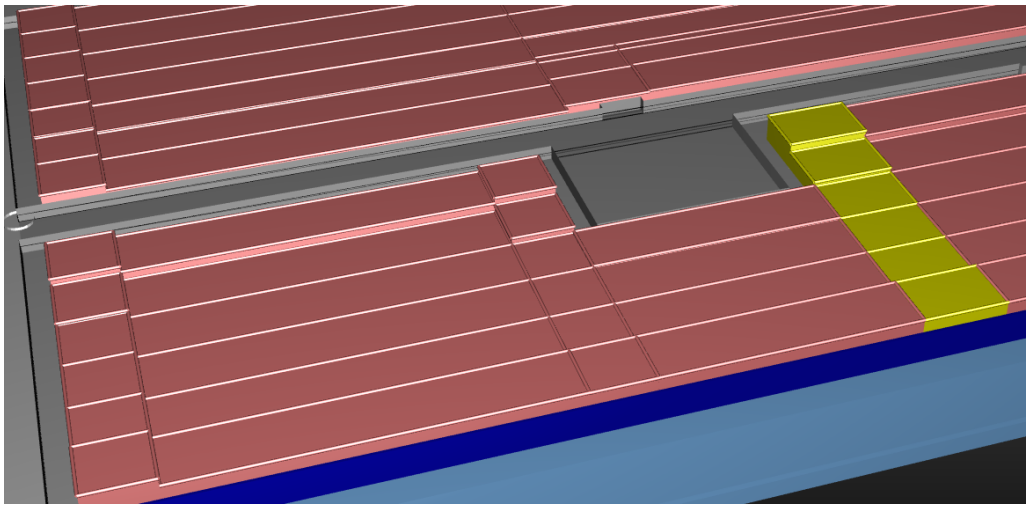

Abbildung 2.4: Ansicht einer Palette mit einem hervorgehobenen Betoniersatz (gelb). Schalungs- und Bewehrungselemente wurden ausgeblendet.

Der gelb eingefärbte Bereich stellt einen einzelnen Betoniersatz dar, der in sechs Blöcke für die einzelnen Schnecken bzw. Klappen aufgeteilt ist. Anhand der Blockhöhe ist zu sehen, dass an den Außenseiten mehr Beton ausgetragen werden soll als in der Mitte, um so während des Rüttelns die Lücke zur Schalung auszufüllen. Dies wird durch das unterschiedliche Austragsverhältnis der Schnecken erreicht, welches der Leitrechner vorgibt.
Aus den Betoniersätzen wird der *Sollwert* berechnet. Dieser gibt an, wie viel Beton pro Weg (in $\frac{g}{mm}$ oder $\frac{kg}{m}$) ausgetragen werden soll und dient der Regelung als Eingangsparameter (siehe Abbildung 4.2). [15, S. 35f]

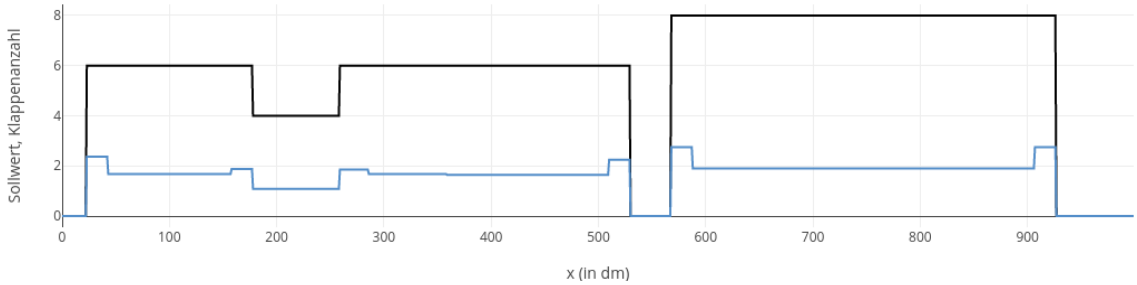

Abbildung 2.5: Auszug einer realen Sollkurve (blau) und der dazugehörigen Klappenanzahl (schwarz).

In Abbildung 2.5 ist die Sollkurve zu den Betoniersätzen abgebildet, die ebenfalls in Abbildung 2.4 dargestellt sind. Die Reduzierung der Klappenanzahl wie auch der höhere Austrag an den Außenrändern in X-Richtung ist erkennbar. An der Stelle, wo keine Klappen geöffnet sind und die Sollkurve den Wert 0 hat, befinden sich Positioniersätze.

Der zweite Parameter für die Regelung ist der *Istwert*, welcher analog zum Sollwert das Verhältnis von Gewicht zu Strecke angibt. Der Istwert wird allerdings mit den gefilterten Sensordaten des Betonverteilers berechnet (siehe Block unten links in Abbildung 4.2). Aufgabe der Regelung ist es, die Regelgrößen *Schneckendrehzahl* und *Fahrgeschwindigkeit* so zu bestimmen, dass die Differenz zwischen Soll- und Istwert möglichst gering ist. Dies ist durch je einen PI-Regler für jede Regelgröße realisiert. [15, S. 37,65]
Die Information welche Klappen geöffnet bzw. welche Schnecken aktiv sein sollen, wird nicht geregelt, sondern anhand der Vorgaben des Leitrechners gesteuert.

Bei der Inbetriebnahme eines Betonverteilers werden zudem einige feste Werte in der Speicherprogrammierbaren Steuerung (SPS) gesetzt. Darunter befindet sich der

Positionswert des Palettenanfangs, die minimale und maximale Drehzahl der Förderschnecken (bzw. Stachelwalze) sowie die Regelungskurven für die Geschwindigkeit des Betonverteilers und der Drehzahl der Förderschnecken (bzw. Stachelwalze) in den PI-Reglern [15, S. 17].

Da Optimierungsmöglichkeiten dieser Regelung mithilfe Neuronaler Netze untersucht werden sollen, gibt der folgende Abschnitt zunächst eine Einführung in die Funktionsweise Neuronaler Netze.

2.2 Neuronale Netze

Ein wesentlicher Aspekt in dem sich Mensch und Maschine unterscheiden, ist, dass ein Prozessor Daten anders verarbeitet als ein Gehirn. Entsprechend werden beispielsweise Rechenaufgaben mit einem Prozessor anders gelöst als in einem Gehirn. [9, S. 3]
Die Forschung im Bereich der Neurologie untersucht die Funktionsweise biologischer Neuronen. Dabei entstanden auch mathematische Modelle für künstliche Neuronale Netze wie beispielsweise von McCulloch und Pitts [34]. Diese Modelle bilden nur Teilaspekte der Realität ab, können aber bereits dazu genutzt werden, um Computer lernfähiger und fehlertoleranter zu machen.
In diesem Abschnitt wird ausgehend von der biologischen Grundlage erläutert, wie einige ausgewählte künstliche Neuronale Netze aufgebaut sind und wie sie trainiert werden.

2.2.1 Biologische Grundlage

Das Vorbild künstlicher Neuronaler Netze ist das zentrale Nervensystem und insbesondere das Gehirn von Menschen oder Tieren. Das Nervensystem von Lebewesen lässt sich in drei Bereiche untergliedern. Es umfasst zum einen im Körper verteilte sensorische Systeme zur Informationsgewinnung, außerdem das zentrale Nervensystem für die Informationsverarbeitung sowie motorische Systeme für die Steuerung von Bewegungen. Die Informationsverarbeitung erfolgt dabei hauptsächlich durch die Neuronen. [30, S. 9] Das menschliche Gehirn enthält schätzungsweise 86 ($\pm$ 8) Milliarden Neuronen und damit mehr als die Gehirne anderer Primaten [4].
Aufgrund der Komplexität dieser biologischen Nervensysteme, beschränkt sich die folgende Beschreibung im Wesentlichen auf die Aspekte, welche auch bei den KNN abgebildet werden.

Abbildung 2.6 zeigt den vereinfachten Aufbau eines solchen Neurons, welches die Grundlage für künstliche Neuronen ist.

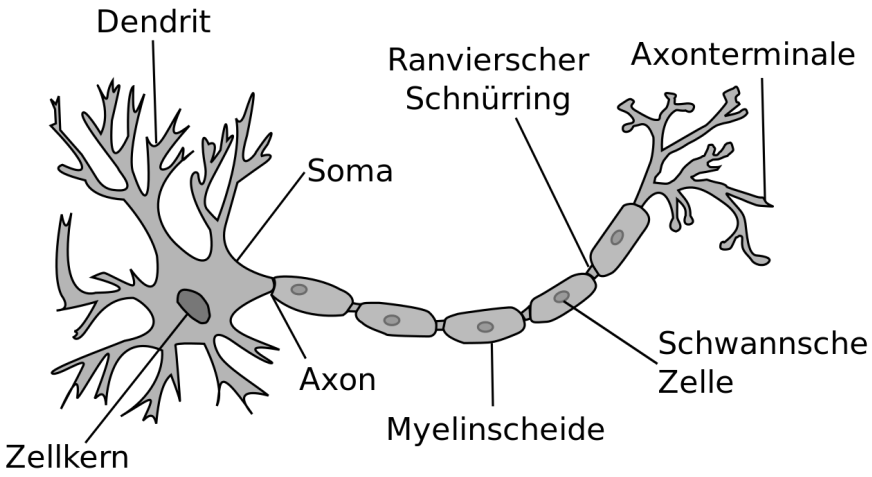

Abbildung 2.6: Vereinfachte Darstellung eines Neurons [56]

Neuronale Netze bestehen aus einer Vielzahl von Neuronen. Neuronen sind Zellen, die elektrische Aktivität akkumulieren und weiterleiten können. An dem Zellkörper (*Soma*) befinden sich die *Dendriten* (linke Seite von Abbildung 2.6) und das *Axon* mit den *Axonterminalen* (rechte Seite). Die Axonterminale führen zu den Dendriten anderer Neuronen, berühren diese aber nicht, sondern bilden den 10nm bis 50nm großen synaptischen Spalt. Auf diese Weise hat ein menschliches Neuron ca. 1000 eingehende und 1000 ausgehende Verbindungen zu anderen Neuronen. Diese Verbindungsstellen werden als *Synapsen* bezeichnet.
Durch die höhere Konzentration von negativ geladenen Ionen an der Innenseite der Zellmembran und positiv geladenen Ionen an der Außenseite, gibt es im Ruhezustand einen Ladungsunterschied (*Aktionspotential*) zwischen dem Neuron und seiner Umgebung von ca. -70mV. Dieser Unterschied kann durch das Ausschütten von Neurotransmittern an den angrenzenden Axonterminalen erhöht (*exzitatorische Verbindung*) oder verringert (*inhibitorische Verbindung*) werden. Die an einem Neuron anliegenden Effekte (erregend und hemmend) summieren sich auf. Übersteigt die Spannung in der Zelle einen Schwellenwert, erzeugt das Neuron selbst einen Impuls, der durch das Axon und die Synapsen an alle verbundenen Neuronen weitergeleitet wird. Die Fortpflanzungsgeschwindigkeit dieses Signals wird durch die *Myelinscheide* beeinflusst. Je stärker ein Axon myelinisiert ist, desto schneller breitet sich das Signal aus. Der Impuls bewirkt zudem die Zurücksetzung des Aktionspotentials auf ca. -70mV. [1, S. 10f] [30, S. 9,10] [29]

Informationen können in Neuronalen Netzen an mehreren Stellen gespeichert werden. Zum einen unterscheiden sich die Aktionspotentiale der Neuronen. Je nachdem, wie groß der grundsätzliche Spannungsunterschied ist, sind höhere oder niedrigere Impulse durch andere Neuronen notwendig, damit das Neuron selbst einen Impuls erzeugt. In Form dieses Schwellenwertes können Informationen kodiert werden. [1, S. 11]
Eine weitere Form des Informationsspeichers ist die Leitfähigkeit der Synapsen, welche sich erhöht, umso mehr Spannungsimpulse übertragen werden. Die elektrische Leitfähigkeit nimmt über die Zeit wieder ab, wenn keine Impulse übertragen werden. [9, S. 9]

Mehrere miteinander verbundene Neuronen können durch Veränderungen dieser genannten Parameter lernen und Informationen speichern. Wie dieses Prinzip auch für Computer nutzbar gemacht werden kann, beschreibt der folgende Abschnitt.

2.2.2 Künstliche Neuronale Netze

Wie bereits in Abschnitt 2.2.1 erwähnt, orientiert sich die Implementierung künstlicher Neuronen an den Grundprinzipien biologischer Neuronen. Der Aufbau und die Funktionsweise eines künstlichen Neurons wird nun anhand der folgenden Abbildung erläutert und in Relation zur biologischen Referenz (siehe 2.2.1) gesetzt.

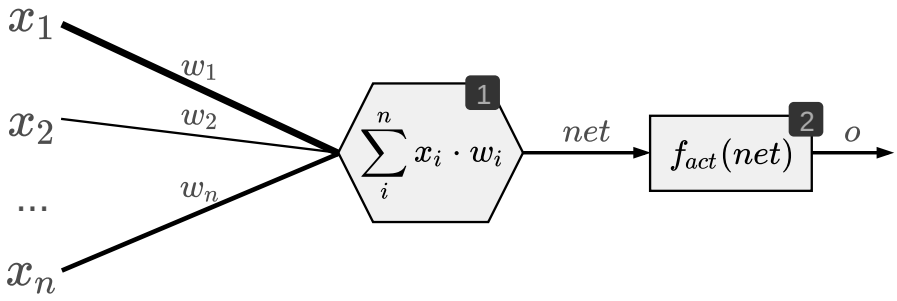

Abbildung 2.7: Aufbau eines künstlichen Neurons

Auf der linken Seite von Abbildung 2.7 sind die Ausgabesignale der vorangestellten Neuronen dargestellt, welche zugleich die *Eingangssignale* $x_i = \{x_1, x_2, ..., x_n\}$ für das aktuelle Neuron sind.
Die biologischen Synapsen werden in Form von *Gewichten* $w_i = \{w_1, w_2, ..., w_n\}$ der Verbindungen zwischen den Neuronen modelliert. Diese Gewichte bestimmen wie stark der Einfluss eines Eingangssignals auf das Neuron ist und repräsentieren somit

auch Informationen. Analog zu der Leitfähigkeit von biologischen Synapsen, können auch die Verbindungsgewichte durch einen Lernvorgang verändert werden.
Um die Akkumulierung der ankommenden elektrochemischen Impulse an biologischen Neuronen zu modellieren, summiert ein künstliches Neuron alle gewichteten Eingangswerte auf (siehe Nr. 1 in Abbildung 2.7). Bei künstlichen Neuronen findet zudem das unterschiedlich hoch ausfallende Aktionspotential Berücksichtigung, welches als Schwellenwert für das Auslösen des Impulses zu verstehen ist. Dafür wird ein *Biasneuron* mit dem konstanten Ausgangswert 1 als Eingabe des aktuellen Neurons hinzugefügt und n entsprechend erhöht, damit es bei der Berechnung der Netzeingabe *net* berücksichtigt wird. Der Schwellenwert entspricht somit dem Verbindungsgewicht. Die *Aktivierungsfkunktion* (2) ist in der Lage, die Weitergabe eines Signals zu verhindern oder zuzulassen. Andere Arten der Aktivierungsfkunktion schwächen Signale ab, ohne sie vollständig zu blockieren. In Abschnitt 2.2.5 sind mehrere dieser Funktionen, sowie deren Vor- und Nachteile, genauer erläutert.

Ein einzelnes Neuron ist nicht in der Lage komplexe Aufgaben zu bewältigen. Dies ändert sich bei der Kombination mehrerer Neuronen. Im nächsten Abschnitt wird erläutert, auf welche Arten Neuronen zu einem Netz verbunden werden können.

2.2.3 Netzstrukturen und -arten

Die Wahl einer geeigneten Netzstruktur ist abhängig von der Problemstellung, die ein Netz lösen soll. In diesem Abschnitt werden einige Netze sowie deren Eignung für bestimmte Aufgaben erläutert. Trainingsmethoden für diese Netze sind in Abschnitt 2.2.4 beschrieben.

Singlelayerperceptron

Das im vorherigen Abschnitt vorgestellte Neuron bzw. neuronale Netz (siehe Abbildung 2.7), ist die einfachste Form eines künstlichen Neuronalen Netzes. Netze, die genau eine Schicht Eingabeneuronen und eine damit verbundene Schicht von Ausgabeneuronen enthalten, werden *einlagiges Perceptron* oder *Singlelayerperceptron* genannt [27, S. 84].

Weil lediglich eine Schicht trainierbarer Gewichte vorhanden ist, kann das einlagige Perceptron allerdings nur lineare Funktionen repräsentieren und ist dadurch auf deutlich weniger Probleme anwendbar, als ein mehrlagiges Netz. Erst mit einem

mehrlagigem Perceptron oder *Multilayerperceptron* können nicht-lineare Funktionen abgebildet werden. [23, S. 533] [27, S. 123ff]

Multilayerperceptron

Welcher Teil eines Netzes genau als Layer oder Schicht bezeichet wird, ist von Autor zu Autor verschieden [27, S. 87]. In dieser Arbeit wird ein n-Layer-Perceptron dadurch definiert, dass es aus n verarbeitenden Neuronenschichten besteht. Das Multilayerperceptron (*mehrlagiges Perceptron*) enthält somit eine nicht-verarbeitende Eingabeschicht, mindestens eine verarbeitende, versteckte Schicht (bzw. *hidden Layer*) und eine verarbeitende Schicht mit Ausgabeneuronen (siehe Abbildung 2.8).

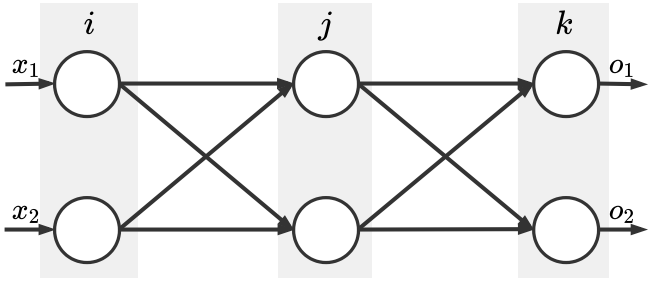

Abbildung 2.8: Ein zweilagiges Multilayerperceptron mit einer versteckten Neuronenschicht.

Der Ausgabevektor einer Neuronenschicht wird wie in der obigen Abbildung auch weiterhin in dieser Arbeit mit o bezeichnet. Die Schichten eines Netzes besitzen Indizes $(i, j, k, ...)$, um Verbindungen sowie Ein- und Ausgänge der Neuronen eindeutig benennen zu können. Somit ist x_i beispielsweise der Eingangsvektor der ersten Schicht, o_k der Ausgabevektor der letzten Schicht und die Matrix w_{ij} enthält alle Verbindungsgewichte zwischen erster und zweiter Neuronenschicht.

Wenn $f_{act}(x)$ die Aktivierungsfunktion ist, gilt für die Ausgabe der versteckten Schicht folgende Gleichung:

$$o_j = f_{act}(x_i \cdot w_{ij}) \tag{2.1}$$

So ergibt sich für die Gesamtausgabe des dargestellten Netzes

$$o_k = f_{act}(o_j \cdot w_{jk}) = f_{act}(f_{act}(x_i \cdot w_{ij}) \cdot w_{jk}). \tag{2.2}$$

Wie G. Cybenko in der Veröffentlichung *Approximation by superpositions of a sigmoidal function* gezeigt hat, kann mit einem solchen zweischichtigen Netz jede

beliebige stetige Funktionen approximiert werden. Voraussetzung dafür ist die Verwendung von nicht-linearen Aktivierungsfunktionen. [7]

Ein Neuronales Netzes kann auch als gerichteter Graph $G = (V, E)$ definiert werden. Neuronen sind dabei als Knoten V und die Verbindungen (Synapsen) als Kanten E dargestellt. Die Richtung der Kanten zeigt außerdem den Signalfluss an.
Mit dieser Definition kann ein Neuronales Netz als *vorwärtsgerichtet* bzw. als *feed-forward*-Netz bezeichnet werden, wenn der Graph des Netzes kreisfrei ist. Das trifft auch auf die beiden bisher betrachteten Netze zu. Vorwärtsgerichtete Netze sind vor allem für Klassifikationsaufgaben geeignet. Dazu gehört beispielsweise auch die Objekterkennung auf Bildern mit *Convolutional Neural Networks*[1] (CNN). Sie erzeugen für die gleichen Eingabedaten immer die gleiche Ausgabe.

Wenn sequenzielle Daten verarbeitet werden sollen bzw. der Ein- oder Ausgabevektor keine feste Größe hat, sind viele *rekurrente (rückgekoppelte) Netze* besser geeignet als vorwärts gerichtete Netze[2]. Dies ist beispielsweise bei einer Übersetzungsaufgabe der Fall, wo eine Wort für Wort Übersetzung keine guten Ergebnisse erzeugt. Vielmehr sollten bei der Übertragung des aktuellen Wortes in eine andere Sprache auch vorhergehende und nachfolgende Ausdrücke berücksichtigt werden. Mit rückgekoppelten Neuronalen Netzen ist dies möglich.
Der Graph eines rekurrenten Netzes ist durch die vorhandenen Rückkopplungen nicht kreisfrei. Die Rückkopplungen werden beispielsweise durch das Zwischenspeichern von Neuronenausgaben und deren Berücksichtigung zu darauffolgenden Zeitpunkten realisiert. So können frühere Netzzustände Einfluss auf die aktuelle Ausgabe haben [17, S. 1]. Daraus ergibt sich auch, dass ein rekurrentes Netz für die gleiche Eingabe zu unterschiedlichen Zeitpunkten auch unterschiedliche Ausgabewerte erzeugen kann.
Im Folgenden werden mehrere Möglichkeiten gezeigt, wie Rückkopplungen in Neuronalen Netzen umgesetzt werden.

[1] siehe [32] oder `https://wiki.tum.de/display/lfdv/Convolutional+Neural+Networks`

[2] Grundsätzlich können aber auch vorwärtsgerichtete Netze sequenzen verarbeiten. Siehe [8]

Hopfield-Netz

Ein Hopfield-Netz besteht aus binären Neuronen V, welche einen Zustand $V_i = -1$ (nicht aktiv) oder $V_i = 1$ (aktiv) einnehmen können[3]. Außerdem hat jedes Neuron einen festen Schwellenwert U_i. Das Netz ist nicht, wie bei dem Multilayerperceptron, in Schichten von Neuronen gegliedert, vielmehr gibt es zwischen allen möglichen Neuronenpaaren ungerichtete, gewichtete Verbindungen T_{ij}. Wobei $T_{ij} = T_{ji}$ ist und es keine Verbindungen von einem Neuron zu sich selbst gibt ($T_{ij} = 0$ wenn $i = j$). Der Graph eines Hopfield-Netzes ist somit nicht kreisfrei (siehe Abbildung 2.9), weswegen dieses Netz als rekurrentes Neuronales Netz bezeichnet wird. [18]

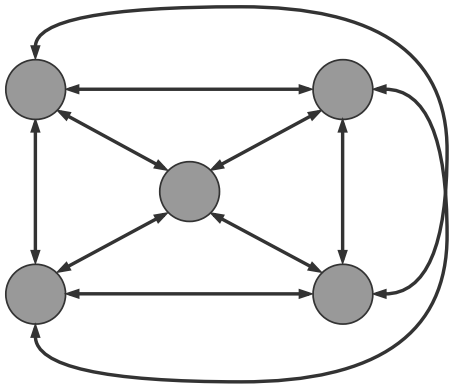

Abbildung 2.9: Ein Hopfieldnetz mit fünf Neuronen.

Hopfield-Netze sind Assoziativspeicher und können sich Zustände merken. Ein gebräuchliches Beispiel ist die Erkennung von Mustern in einem Pixelfeld. Anders als bei dem Multilayerperceptron gibt es keine Eingabe- oder Ausgabeschicht, sondern alle Neuronen des Netzes fungieren als Ein- und Ausgabeneuronen. Wenn Muster beispielsweise von einem Pixelfeld gelernt werden sollen, dann entspricht die Anzahl der Neuronen n auch der Anzahl der Pixel. Nach dem Lernvorgang (siehe 2.2.4 Lernverfahren/Training), können neue (dem Netz unbekannte) Muster x angelegt werden. Daraufhin aktualisieren sich die Neuronenzustände asynchron nach der folgenden Regel (Gleichung 2.3). [9, S. 271f]

$$V_i = \begin{cases} -1 & falls \quad \sum_{\substack{j=1 \\ j \neq i}}^{n} T_{ij} V_j < U_i \\ 1 & sonst \end{cases} \tag{2.3}$$

[3]Ursprünglich wurde das Netz mit den Zuständen 0 und 1 vorgestellt [18], jedoch hat die später verwendete Variante mit -1 und 1 mehrere Vorteile und wird deswegen in dieser Arbeit verwendet. [9, S. 271] [27, S. 136]

Von einem zufällig gewählten Neuron, werden alle anliegenden Signale (Zustand eines Verbundenen Neurons $V_j \cdot$ Verbindungsgewicht T_{ij}) aufsummiert. Der neue Zustand des Neurons ist -1, wenn diese Summe kleiner ist als der Schwellenwert U_i und 1, wenn die Summe größer oder gleich U_i ist. Der Aktualisierungsprozess wird so oft wiederholt, bis das Netz einen stabilen Zustand erreicht, d.h. bis sich kein Neuronenzustand mehr ändert[4].
Neue Muster, die den gelernten Mustern ähneln, lassen sich durch dieses Verfahren in gelernte Muster umwandeln und können so erkannt werden. [9, S. 272f]

Hopfield-Netze nehmen immer einen stabilen Zustand ein, allerdings kann es vorkommen, dass dies nicht einer der gelernten Zustände ist [9, S. 277] [30, S. 123]. Eine weitere Einschränkung ergibt sich, da die Größe des Netzes und damit auch die Speicherkapazität von der Größe der Eingabe abhängig ist. Je nachdem, wie viele Muster gelernt werden und wie hoch die Fehlertoleranz sein soll, kann die Kapazität für die Wiedererkennung aller Muster zu gering sein [9, S. 274].
Da das Einschwingen je nach Muster unterschiedlich lang dauert, kann dies ebenfalls ein Nachteil gegenüber Netzen sein, welche die Daten in eine Funktion mit konstanter Laufzeit überführen (z.B. MLP, Jordan, ...) [9, S. 271]. Der Vorteil eines Hopfield-Netzes ist hingegen, dass die Verbindungsgewichte bei dem Lernvorgang nur einmal berechnet werden müssen (siehe 2.2.4 Lernverfahren/Training).

Hopfield-Netze hatten in den 80er Jahren einen großen Einfluss auf die Entwicklungen Neuronaler Netze, mittlerweile haben jedoch die in den nächsten Abschnitten beschriebenen Netze eine größere Bedeutung [30, S. 141].

Jordan Netz

M. I. Jordan stellte 1986 ein rekurrentes Netz vor, um damit sequentielle Eingaben zu verarbeiten. Die Grundlage bildet ein Multilayerperceptron mit einer internen Neuronenschicht.

[4] anschauliches Beispiel: `https://www.coursera.org/learn/neural-networks/lecture/9ZOr2/hopfield-nets-13-min` [abgerufen: 6.3.2018]

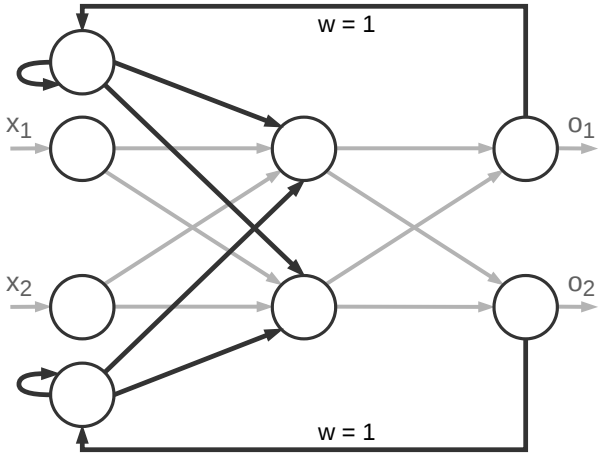

Abbildung 2.10: Struktur eines mehrschichtigen, rückgekoppelten Netzes nach JORDAN.

Für jedes Neuron in der Outputschicht gibt es ein verbundenes *Kontextneuron*, dessen Ausgang mit der ersten Schicht nach den Inputneuronen voll verbunden ist (vgl. Abbildung 2.10). Kontextneurone geben ihren Eingangswert nicht direkt, sondern erst in dem nächsten Zeitschritt weiter und dienen somit als Zwischenspeicher. Jedes Kontextneuron besitzt außerdem eine Rückkopplung auf sich selbst, indem sein Ausgang als zweiter Eingabewert für den folgenden Zeitschritt verwendet wird. Bei vielen Anwendungen wird dieser Aspekt aber nicht implementiert [27, S. 175]. Die Verbindungsgewichte zwischen Outputneuron und Kontextneuron sind auf den Wert 1 festgelegt und werden auch nicht durch Training angepasst. [23, S. 534f]

Wenn die Gewichte zwischen den Kontextneuronen und der versteckten Schicht auf 0 gesetzt werden, reduziert sich das rückgekoppelte Netz auf ein Multilayerperceptron. Dies zeigt, dass rekurrente Netze komplexer sind als vorwärtsgerichtete Netze. [27]

Die Komplexität der Jordan-Architektur kann nicht beliebig erhöht werden, da die Größe der Ausgabeschicht und damit auch die Anzahl der rekurrenten Verbindungen durch die Aufgabenstellung festgelegt ist. [31, S. 255]
Um diesem Problem zu begegnen, stellte ELMAN 1990 ein leicht abgewandeltes Netz vor.

Elman-Netz

Elman bezieht sich in seiner Veröffentlichung *Finding Structure in Time* auf das Jordan-Netz, wandelt diesen Ansatz allerdings etwas ab. Anstatt jedes Outputneu-

ron an ein Kontextneuron zu koppeln, schlug ELMAN vor, die Neuronen der internen Schicht mit je einem Kontextneuron zu verbinden (vgl. Abbildung 2.11). Diese Verbindungen werden zudem nicht trainiert, sondern haben das feste Gewicht von 1. Jedes Kontextneuron ist wie bei einem Jordan-Netz mit jedem Neuron der versteckten Schicht verbunden. Diese Verbindungen sind auch trainierbar. [8]
Die folgende Abbildung veranschaulicht das beschriebene Konzept.

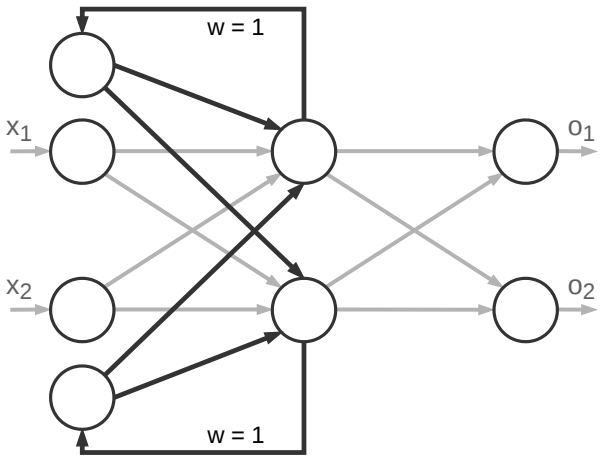

Abbildung 2.11: Struktur eines mehrschichtigen, rückgekoppelten Netzes nach ELMAN. [8]

In der ursprünglichen Veröffentlichung von Elman gibt es keine direkten Rückkopplungen der Kontextneuronen zu sich selbst wie bei einem Jordan Netz.
In der Literatur sind allerdings auch Abwandlungen zu finden. D. KRIESEL definiert Elman-Netze z.B. so, dass jede verarbeitende Schicht des Netzes auch Kontextneuronen besitzt [27] (weitere Abwandlung: [31, S. 257]).
Fest steht, dass mit dieser Netzstruktur die Komplexität der Rückkopplungen nicht mehr von der Anzahl der Outputneuronen abhängig ist, welche im Gegensatz zu den versteckten Neuronen durch die Aufgabenstellung bestimmt wird [31, S. 257].

Die Jordan- und Elman-Netze, können Daten aus vorherigen Eingabewerten bzw. Zuständen für die aktuelle Ausgabe berücksichtigen. In der Praxis ist das allerdings nicht für beliebig große Abstände zwischen den verwandten Daten möglich [16]. Der Grund dafür ist das *vanishing gradient*-Problem, welches im Abschnitt 2.2.5 erläutert wird.
Um dieses Problem zu umgehen, wurden LSTM-Netze entwickelt.

LSTM-Netze

Die 1997 von HOCHREITER und SCHMIDHUBER vorgestellten LSTM-Netze umgehen das Vanishing Gradient Problem, indem sie die zu berücksichtigenden Daten einschränken. Eine LSTM-Zelle kann z.B. anstelle eines Neurons in ein Neuronales Netz integriert werden. Diese Zelle besitzt neben den Eingängen x und dem Ausgang h (o hat in diesem Abschnitt eine andere Bedeutung) mit dem *cell state* C_{t-1} und der Ausgabe des vorherigen Schrittes h_{t-1} noch zwei weitere Eingänge.
Mit dem aktuellen Input x_t zum Zeitpunkt t und dem Ergebnis des vorherigen Schrittes h_{t-1} wird innerhalb der LSTM-Zelle der cell state C_{t-1} verändert sowie die Ausgabe h_t berechnet. In der folgenden Abbildung (2.12) ist neben den Ein- und Ausgabewerten auch der innere Aufbau einer LSTM-Zelle dargestellt. [17] [36]

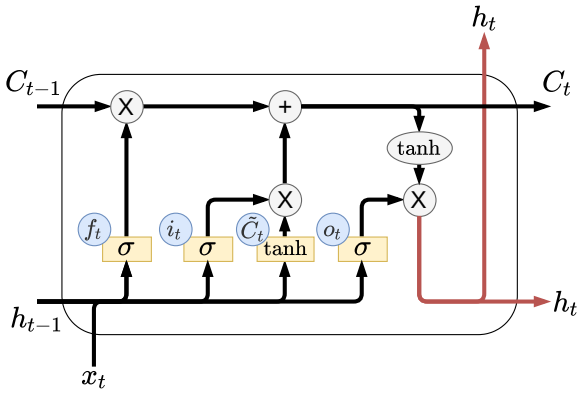

Abbildung 2.12: Aufbau einer LSTM-Zelle nach OLAH [36]

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \quad (2.4)$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \quad (2.5)$$

$$o_t = \sigma(W_s \cdot [h_{t-1}, x_t] + b_s) \quad (2.6)$$

$$\tilde{C}_t = tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \quad (2.7)$$

Kleine Neuronale Netze innerhalb der Zelle, die als *gates* bezeichnet werden, ermöglichen es, relevante Langzeitabhänigkeiten zu behalten und irrelevante zu verwerfen. In einer LSTM-Zelle gibt es ein *input gate* (i), ein *forget gate* (f) und ein *output gate* (o) [17].
Das forget gate f_t (vgl. Abbildung 2.12) berechnet nach Gleichung 2.4 einen Vektor, der so groß ist wie der *cell state* und durch die Anwendung der Logistikfunktion[5] Werte zwischen 0 und 1 enthält. W steht für die Gewichtsmatrix des jeweiligen gates und b für den Biasvektor. Durch die elementweise Multiplikation von f_t mit dem vorherigen Zellzustand C_{t-1}, wird dessen Einfluss abgeschwächt bzw. ein Teil des Zustands „vergessen“. [36]

[5] siehe Abschnitt 2.2.5

Das zweite gate (input gate i_t) wird darauf trainiert, Informationen aus den Eingabedaten (h_{t-1} und x_t) auszuwählen, die in den neuen Zellzustand C_t einfließen sollen. Auch dieses gate gibt einen Vektor mit Werten zwischen 0 und 1 aus (siehe Gleichung 2.5), der mit $\tilde{C}_t$ (Gleichung 2.7) multipliziert und zu dem neuen Cell State C_t addiert wird. Zusammengefasst gilt für den aktualisierten Zellzustand folgende Gleichung [36].

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \tag{2.8}$$

Das output gate o_t bestimmt analog zu den anderen gates, welche Teile des Zellzustands von der LSTM-Zelle ausgegeben werden. Die Berechung des Ausgabevektors h_t erfolgt schließlich nach Gleichung 2.9 [36].

$$h_t = o_t \cdot tanh(C_t) \tag{2.9}$$

Viele Erfolge Neuronaler Netze im Bereich der Sequenzverarbeitung gehen auf LSTM-Netze zurück [36]. Wie auch schon HOCHREITER und SCHMIDHUBER gezeigt haben, lernen LSTM-Netze zudem schneller als RNNs und haben eine höhere Erfolgsrate bei der Mustererkennung in Sequenzen wie beispielsweise bei dem Lernen der Reber Grammatik [17].
LSTM-Zellen können beispielsweise anstelle der versteckten Neuronen eines Multilayerperceptrons eingesetzt werden, um dieses zu einem rekurrenten Netz zu machen. Trainiert werden LSTM-Netze durch den *Backpropagation Trough Time*-Algorithmus (siehe Abschnitt 2.2.4).

Bestimmen der Netzgröße

Um die optimale Konfiguration eines Neuronalen Netzes zu ermitteln, gibt es kein allgemeingültiges Verfahren [31, S. 247]. Es können lediglich einige grobe Rahmenbedingungen helfen, die Auswahl einzuschränken.

So gibt es für die Wahl der Neuronenanzahl pro Schicht Erfahrungswerte aber keine allgemeingültige Formel. JEFF HEATON schlägt beispielsweise eine der folgenden drei Regeln vor, die als Ausgangspunkt für Versuche mit verschiedenen Neuronenzahlen pro Schicht dienen können [13] [30, S. 76]:

- N_{hidden} = zwischen N_{in} und N_{out}
- $N_{hidden} = \frac{2}{3}N_{in} + N_{out}$
- $N_{hidden} < 2N_{in}$

Ziel ist es, dass ein Netz so viele Neuronen wie nötig und so wenige wie möglich besitzt. Bei zu wenigen Neuronen ist das Netz nicht in der Lage, genügend Informationen zu speichern, um die Aufgabe zu lösen. Ist ein Netz zu groß, wird das Training aufgrund der vielen Verbindungen aufwändiger. Andererseits besteht die Gefahr, dass das Netz die Trainingsbeispiele auswendig lernt und nicht mehr generalisieren kann. D. KRIESEL schlägt vor: *„zunächst mit wenigen Neuronen zu trainieren und so lange neue Netze mit mehr Neuronen zu trainieren, wie das Ergebnis noch signifikant verbessert und vor allem die Generalisierungsleistung nicht beeinträchtigt wird [...]“*. [27, S. 58f,103f]

Um nicht manuell verschiedene Konfigurationen erstellen, trainieren und testen zu müssen, kann dieser Vorgang auch automatisiert werden. Eine Vorgehensweise, die sich für eine solche Aufgabe eignet und verwendet wird, ist der Einsatz von genetischen Algorithmen [48].
Ein genetischer Algorithmus orientiert sich an der Evolutionstheorie und erzeugt zunächst eine *Population* von zufälligen aber validen Lösungskandidaten (*Individuen*). In diesem Fall wären das Neuronale Netze mit verschiedenen Konfigurationen. Diese Netze werden anschließend trainiert, um den Fehler für ein bestimmtes Trainingsset als Vergleichskriterium (*Fitness*) verwenden zu können. Die besten Lösungen pflanzen sich fort, indem sie mit anderen kombiniert werden (*Rekombination*) und bilden damit eine neue *Generation*. Außerdem können durch zufällige *Mutationen* weitere neue Konfigurationen gebildet werden. Der Prozess von Rekombination, Mutation und Bewertung wird so oft wiederholt, bis eine zufriedenstellende Lösung gefunden wurde. [30, S. 167-170]
Genetische Algorithmen können somit helfen, geeignete Parameter für ein Neuronales Netz zu finden. Vor allem wenn die Anzahl der Möglichkeiten und damit der Suchraum sehr groß ist, kann es sich lohnen einen solchen Algorithmus zu verwenden.
Diese Beschreibung umfasst nur einige der wesentlichen Aspekte genetischer Algorithmen. Ausführlichere Informationen sind beispielsweise in *Computational Intelligence* von R. KRUSE et al. ab Seite 156 zu finden.

Nach der Beschreibung unterschiedlicher Arten und Funktionsweisen Neuronaler Netze wurde noch nicht erläutert, wie diese trainiert werden können. Es folgt nun die Vorstellung der grundlegendsten Verfahren.

2.2.4 Lernverfahren/Training

Eine Vielzahl an Fähigkeiten, wie z.B. das Sprechen muss ein Mensch erst lernen. Biologische Neuronale Netze lernen unter anderem durch die Herstellung, Stärkung, Schwächung oder den Abbau neuronaler Verbindungen [1].
Ebenso ist es notwendig, künstliche Neuronale Netze zu trainieren. Dabei ist im Bereich der KI vor allem das Lernen von Konzepten Gegenstand der Forschung und weniger das Auswendiglernen, da dies Computer bereits gut umsetzen können. [9, S. 192]

Eine Art des Lernens ist das *überwachte Lernen* (Lernen mit Lehrer). Dabei wird ein System anhand von Beispielen und den dazugehörigen Lösungen trainiert. Eine Anwendung davon sind Klassifizierungsaufgaben, wie das Bestimmen von Blumen anhand ihrer Merkmale. Einem Neuronalen Netz werden dafür als Eingabedaten die Merkmale einer Pflanze übergeben, das Netz berechnet eine Ausgabe und vergleicht diese mit der richtigen Lösung. Anhand des gemachten Fehlers kann das Netz schließlich angepasst werden und lernen. Neben Neuronalen Netzen gibt es noch andere Algorithmen, die auf diese Weise lernen (z.B. k-Nearest-Neighbour). [9, S. 257f]

Clusteringverfahren können Daten anhand ihrer Merkmale in Gruppen aufteilen, ohne Beispieldaten als Referenz zu haben. Diese Art des Lernens wird deswegen als *unüberwachtes Lernen* (Lernen ohne Lehrer) bezeichnet. Neben Neuronalen Netzen ist z.B. der k-Means[6] Algorithmus in der Lage Daten zu clustern.

Die letzte hier erwähnte Lernform ist das *Lernen durch Verstärkung* (*Reinforcement Learning*). Es ähnelt dem überwachten Lernen, mit dem Unterschied, dass das lernende System nicht für jede Eingabe die richtige Lösung kennt. Es bekommt nur vereinzelt positives oder negatives Feedback durch seine Umgebung. [9, S. 257]

In dieser Arbeit steht das überwachte Lernen für Neuronale Netze im Mittelpunkt. Wie bei natürlichen Gehirnen gibt es auch bei KNNs mehrere Anpassungsmöglichkeiten. Dazu gehört das Verändern der Schwellenwerte (bzw. der Verbindung zu den Bias-Neuronen) oder das Entfernen bzw. Hinzufügen von Neuronen und Verbindungen. Die am häufigsten verwendete Methode ist die Anpassung der Verbindungsgewichte, weil sich damit zusätzlich auch der Auf- und Abbau von Verbindungen modellieren lässt. Wenn zunächst alle Neuronen miteinander verbunden sind, kön-

[6] siehe [9, S. 244ff]

nen bei einem Lernprozess beispielsweise nicht benötigte Verbindungen mit Null gewichtet und somit entfernt werden. [27, S. 53]
Je nach Netzart und Anwendungsfall bieten sich verschiedene Lernmethoden an, welche in diesem Unterkapitel erläutert werden.

Für überwachte Lernverfahren von Neuronalen Netzes gibt es verschiedene Abläufe in Bezug auf den Zeitpunkt, zu dem die Gewichte angepasst werden.
Die erste Möglichkeit besteht darin, für jeden einzelnen Eingabevektor die Ausgabe des Netzes sowie den Fehler zu berechnen und die Verbindungsgewichte zu verändern. Diese Vorgehensweise wird als **online learning** (Online-Lernen) bezeichnet. [30, S. 23]. P. Werbos unterscheidet aber beispielsweise noch zwischen *real-time learning*, wo der Trainingsdatensatz nur einmal trainiert wird und *pattern learning*, bei welchem es mehrere Durchläufe durch den Datensatz gibt [55, S. 1553f].

Das **batch/offline learning** unterscheidet sich von den Online-Verfahren, weil dabei eine Gewichtsänderung erst nach der Bearbeitung des gesamten Trainingsdatensatzes durchgeführt wird [55, S. 1553] [30, S. 24].

Ein weiteres online-Verfahren ist das *stochastic learning*, welches bei jeder Iteration zufällig ein Beispiel aus dem Datensatz auswählt.

Hebb-Regel

Die Hebb-Regel modelliert den Aspekt der natürlichen Neuronalen Netze, dass die Synapsen gestärkt werden, je mehr Spannungsimpulse sie übertragen (siehe Abschnitt 2.2.1).

D. Hebb veröffentlichte 1949 die folgende Annahme:
Wenn eine Nervenzelle A wiederholt dazu beiträgt, dass eine andere Zelle B einen Impuls weitergibt (feuert), wird die Verbindung der beiden Zellen gestärkt. Dies führt dazu, dass Zelle B von A noch leichter stimuliert werden kann. Hebb vermutete, dass sich Synapsen ausbilden oder deren Kontaktfläche vergrößern, wenn zwei beieinanderliegende Nervenzellen gleichzeitig aktiv sind. [14, S. 62f]

Die Hebb-Regel findet leicht verändert Anwendung beim Trainieren von Hopfield-Netzen. Wenn ein Hopfield-Netz eine Menge von N Mustern lernen soll, werden die

Kantengewichte wie folgt verändert. Jedes Muster besteht aus einem Vektor p mit n Werten.

$$T_{ij} = \frac{1}{n} \sum_{k=1}^{N} p_i^k p_j^k \tag{2.10}$$

Wenn ein binäres Muster die Werte -1 für inaktiv und $+1$ aktiv enthält, ergibt sich aus Gleichung 2.10, dass bei Gleichheit von p_i^k und p_j^k (gleichzeitig aktiv/nicht aktiv) der Einfluss auf das Kantengewicht positiv und sonst negativ ist. Nachdem alle Gewichte nach dieser Gleichung berechnet wurden, ist das Netz dazu in der Lage, Ähnlichkeiten zwischen neuen Mustern und den gelernten zu erkennen (siehe Abschnitt 2.2.3). Der Vorteil dieser Trainingsmethode ist, dass der Wert jeder Kante nur einmal berechnet werden muss, um das Netz zu trainieren. Lange Trainingszyklen wie bei der Backpropagation (siehe unten) gibt es nicht. [9, S. 271ff]

Delta-Regel

Die Delta-Regel ist ein überwachtes Lernverfahren und beschreibt, wie die Verbindungsgewichte w_i eines Singlelayerperceptrons (2.2.2) verändert werden können, um ein Minimum der Fehlerfunktion zu erreichen.
Sei x der Eingabevektor und y der vorgegebene Ausgabevektor eines Trainingsbeispiels sowie o der Ausgabevektor des Netzes, dann ergibt sich nach der Delta-Regel folgende Gewichtsänderung:

$$\Delta w_i = \alpha \sum_{p=1}^{N} (y^p - o^p) x_i^p \tag{2.11}$$

Wobei α die Lernrate und N die Anzahl der Trainingsbeispiele ist.
Um die Gewichtsänderung zu berechnen, wird zunächst für jeden Trainingsdatensatz der Fehler mit dem Eingangswert multipliziert. Diese Werte ergeben in der Summe die Gewichtsänderung. Damit das Gewicht nicht zu stark angepasst und das gesuchte Minimum übersprungen wird, gibt es die Lernrate, um die Änderung zu skalieren. Dieser Vorgang wird so oft wiederholt, bis die Gewichtsänderungen gegen Null konvergieren.

Die Delta-Regel ist die Grundlage für den *Backpropagation*-Algorithmus, welcher auch mit mehrschichtigen Netzen funktioniert.

Backpropagation

Error Backpropagation bzw. *Fehlerrückführung* ist ein *gradientenbasiertes Lernverfahren* (gradient-based learning method) für Neuronale Netze [33, S. 1]. Es wurde 1974 von P. Werbos vorgestellt, aber vor allem durch eine Veröffentlichung von Rumelhart, Hinton und Williams bekannt [54] [40].

Das Verfahren wird in diesem Abschnitt anhand des Multilayerperceptrons mit einem überwachten Lernprozess erläutert. Ziel dieses Lernverfahrens ist es, die Verbindungsgewichte zwischen den Neuronen so anzupassen, dass die Ausgabe des Neuronalen Netzes hinreichend genau mit der gewünschten Ausgabe übereinstimmt. Neben den Eingabewerten enthalten die Trainingsdaten somit auch die zu erreichenden Ausgabewerte. Das Backpropagation-Verfahren umfasst die folgenden Schritte. [55]

1. Ausgabe des Netzes berechnen Zu Beginn ist es notwendig, für die Eingabe x die Ausgabe o des Neuronalen Netzes zu berechnen.

Bei dem Beispielnetz aus Abbildung 2.13 entspricht die Berechnung der des Multilayerperceptrons. Als Aktivierungsfunktion kommt die Logistikfunktion σ (siehe Abschnitt 2.2.5) zum Einsatz und die Verbindungsgewichte wurden zufällig initialisiert. Wie in der Abbildung ersichtlich, gibt das Netz bei $x = \{1, 1\}$ den Wert $o = \{1\}$ aus.

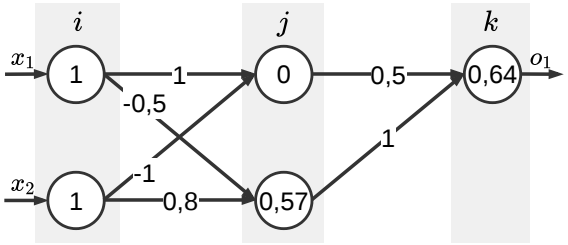

Abbildung 2.13: Multilayerperceptron mit logistischer Aktivierungsfunktion

2. Fehler berechnen Um die Fehlerrückführung zu demonstrieren, soll das Netz in Abbildung 2.13 durch Beispieldaten so trainiert werden, dass es die XOR-Funktion abbildet. Bei 0 = *false* und 1 = *true* ist der Sollwert definiert als:

$$y_1 = (x_1 \vee \neg x_2) \vee (\neg x_1 \wedge x_2). \tag{2.12}$$

Die Wahl der Fehlerfunktion ist von dem Problem/der Aufgabe abhängig. Sie gibt die Abweichung des Sollwertes von der tatsächlichen Ausgabe an und ist im einfachsten Fall die Differenz dieser beiden Werte. Bei mehreren Ausgabewerten könnten sich die Fehler allerdings aufheben, weswegen in diesen Fällen z.B. der *mittlere quadratische*

Fehler (Gleichung 2.13) besser geeignet ist. Der Faktor vor der Summe dient dazu, die Ableitung zu vereinfachen.

$$E = \frac{1}{2}\sum_{i=1}^{N}(y_i - o_i)^2 \tag{2.13}$$

Wobei y_i für den Sollwert und o_i für die tatsächliche Ausgabe eines Neurons steht. Für das Beispiel in Abbildung 2.13 ergibt sich nach dieser Formel der Fehler $E = \frac{1}{2}(y_1 - o_1)^2 = 0,5$. Damit das Netz annähernd die gewünschte Funktion abbildet, muss dieser Fehler minimiert werden.

3. Partielle Ableitungen bilden und Änderung berechnen Um den Fehler zu minimieren, sollen die Verbindungsgewichte des Netzes angepasst werden. Dafür wird die partielle Ableitung des Fehlers nach den Kantengewichten berechnet. Daraus ergibt sich der Betrag und die Richtung der Änderung. Eine Voraussetzung für den Einsatz von Backpropagation ist somit die Verwendung differenzierbarer Aktivierungsfunktionen [27, S. 89].
Da zunächst nur der Fehler an den Ausgabeneuronen bekannt ist, wird auch bei dieser Schicht mit der Fehlerrückführung begonnen. Die Anpassung der Gewichte zwischen versteckter Schicht j und Ausgabeschicht k wird nach der folgenden Formel berechnet [31, S. 219]:

$$\Delta w_{jk} = -\alpha\frac{\delta E}{\delta w_{jk}} \tag{2.14}$$

Die Lernrate α bestimmt den Einfluss der Änderung auf das Kantengewicht und das negative Vorzeichen führt dazu, dass das Gewicht entgegengesetzt zu dem Kurvenanstieg (Richtung Minimum) angepasst wird [31, S. 219]. Da der Fehler von dem Ausgabewert o abhängt und dieser durch die Eingabewerte und Gewichte w bestimmt wird, gilt [38, S. 83]:

$$\frac{\delta E}{\delta w_{jk}} = \frac{\delta E}{\delta o_k}\frac{\delta o_k}{\delta w_{jk}} = \frac{\delta}{\delta o_k}\ \frac{1}{2}\sum_k (y_k - o_k)^2 \cdot \frac{\delta}{\delta w_{jk}}\ \sigma\left(\sum_j w_{jk}\cdot o_j\right) \tag{2.15}$$

Durch Ableiten des Fehlers und der Ausgabe ergibt sich für die letzte Schicht:

$$\frac{\delta E}{\delta w_{jk}} = \delta_k \cdot o_j = (y_k - o_k)\cdot o_k(1 - o_k)\cdot o_j \tag{2.16}$$

Ab der vorletzten Schicht muss zunächst der anliegende Fehler berechnet werden. Dazu wird der oben (Gleichung 2.16) berechnete Fehler aller ausgehenden Verbin-

dungen eines Neurons aufsummiert und anschließend auf die eingehenden Verbindungen aufgeteilt:

$$\frac{\delta E}{\delta w_{ij}} = \delta_j \cdot o_i = \sum_k (\delta_k \cdot w_{jk}) \cdot o_j(1 - o_j) \cdot o_i \tag{2.17}$$

Die Zwischenschritte und eine vollständige Herleitung ist beispielsweise aus dem Buch *Künstliche Intelligenz* von Lämmel und Cleve zu entnehmen [31, S. 219ff].

Durch Anwendung der Lernrate nach Gleichung 2.14 auf die beiden letzten Gleichungen (2.16 oder 2.17) lässt sich die gesuchte Gewichtsänderung (Δw_{jk} bzw. Δw_{ij}) berechnen.

Wie auch bei der *forward-propagation* ist es durch die Verwendung von Matrizen und Vektoren möglich, die Gewichtsänderungen für alle Gewichte einer Verbindungsschicht mit einem Ausdruck zu berechnen [38, S. 87]. Für das obige Beispiel ergeben sich folgende Gewichtsänderungen:

$$\begin{aligned} \Delta w_{jk} &= \alpha \cdot (0 - 0,64) \cdot 0,64(1 - 0,64) \cdot \begin{pmatrix} 0 \\ 0,57 \end{pmatrix} \\ &= \alpha \cdot -0,15 \cdot \begin{pmatrix} 0 \\ 0,57 \end{pmatrix} \\ &= \alpha \cdot \begin{pmatrix} 0 \\ -0,08 \end{pmatrix} \end{aligned} \tag{2.18}$$

4. Verbindungsgewichte anpassen Das Anpassen der Gewichte erfolgt dann für jede Schicht durch die Addition der berechneten Änderung:

$$w_{jk} = w_{jk} + \Delta w_{jk} \tag{2.19}$$

Dieser Vorgang mit den vier vorgestellten Schritten, wird so oft wiederholt, bis die Gewichte konvergieren bzw. der Fehler hinreichend klein ist.
Wie bei der Delta-Regel erwähnt, können auch hier die Gewichte zu unterschiedlichen Zeitpunkten angepasst werden (online/batch). Welche dieser Lernmethode geeignet ist, hängt von der Aufgabenstellung ab.

Backpropagation ist ein sehr verbreitetes Lernverfahren für Multilayerperceptrons, kann aber z.B. für rekurrente Netze nicht ohne weiteres verwendet werden. Allerdings ist der im Folgenden beschriebene *Backpropagation Through Time*-Algorithmus (BPTT) dazu in der Lage.

Backpropagation Through Time (BPTT)

Damit Backpropagation auch für rekurrente Neuronale Netze eingesetzt werden kann, stellte P. WERBOS 1990 einen Ansatz vor, der als *Backpropagation Through Time* bezeichnet wird. [55]
Die Grundidee ist dabei, dass ein rekurrentes Netz als eine Menge von MLPs interpretiert werden kann, die miteinander verbunden sind. Jeder Zeitschritt wird als ein Multilayerperceptron dargestellt. Die folgende Abbildung verdeutlicht diese zeitliche Entfaltung (*unfolding in time*) eines Jordan-Netzes, wo die Ausgänge des vorherigen Zeitpunktes $t-1$ mit der versteckten Neuronenschicht zu dem aktuellen Zeitpunkt t verbunden sind.

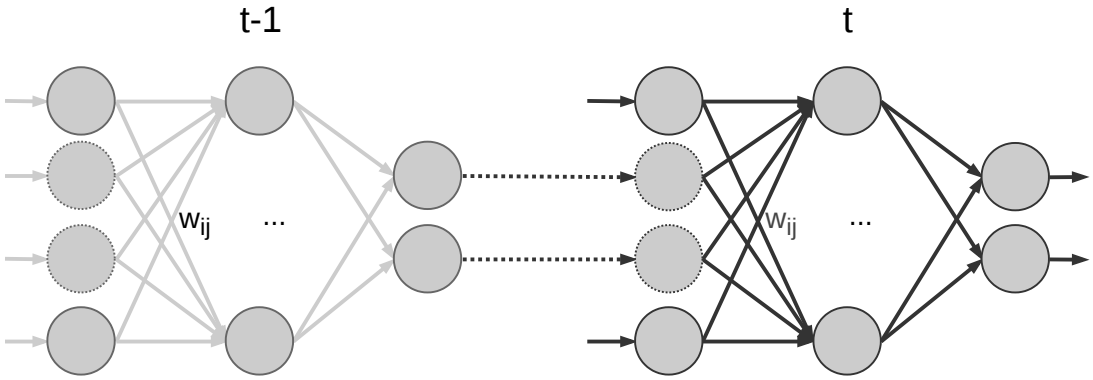

Abbildung 2.14: Ein zeitlich aufgefaltetes Jordan-Netz.

Die Kontextneuronen des Jordan-Netzes (Abbildung 2.14) sind durch unterbrochene Außenlinien gekennzeichnet. Die zeitliche Verknüpfung der Netze stellen die gestrichelten Verbindungen dar, welche mit dem Wert 1 gewichtet sind.

Diese Verkettung von Netzen kann somit als ein großes mehrschichtiges Netz angesehen werden, wo das Training mit Backpropagation möglich ist. Zu beachten ist allerdings, dass alle Gewichte mehrmals, aber zu unterschiedlichen Zeitpunkten vorkommen. Beispielsweise werden für das Gewicht w_{ij} dadurch mehrere Änderungswerte berechnet.
Möglichkeiten diese zusammenzuführen sind beispielsweise der Durchschnittswert oder ein gewichteter Durchschnitt, bei dem Änderungen zunehmend weniger Einfluss haben, je weiter sie in der Vergangenheit liegen. [27]
Backpropagation hat die Eigenschaft, dass die Änderungen mit zunehmender Tiefe des Netzes aus Sicht der Ausgabeschicht immer geringer werden (*vanishing gradient*) [27, S. 95]. Dies liegt zu einem großen Teil an der verwendeten Aktivierungsfunktion

und deren Einfluss auf die Ableitung der Fehlerfunktion. Die Ableitungen sigmoider Funktionen nähern sich in positiver und negativer x-Richtung recht schnell dem Wert null an und haben ihr Maximum bei $x = 0$ (siehe Abschnitt 2.2.5). Da der Wertebereich der Ableitung für die beiden vorgestellten Sigmoidfunktionen zwischen null und eins liegt, geht auch die Ableitung der Fehlerfunktion mit jeder zusätzlichen Schicht immer mehr gegen null. Problematisch ist neben den geringen Gewichtsanpassungen auch, dass Computer Zahlen nicht mit beliebig vielen Nachkommastellen darstellen können. [24]

Dieses Problem kann für rekurrente Netze durch eine Begrenzung der Anzahl von betrachteten Zeitschritten umgangen werden. Allerdings schränkt das auch die Leistungsfähigkeit des Netzes in Bezug auf die Erkennung von Langzeitabhänigkeiten ein. Abhilfe schafft da der Einsatz von LSTMs (siehe Abschnitt 2.2.3).

2.2.5 Aktivierungsfunktionen

Wie bereits erwähnt, gibt es verschiedene Aktivierungsfunktionen. Abhängig von dem konkreten Anwendungsfall, sind manche Funktionen besser geeignet als andere. Außerdem gibt es Neuronale Netze, bei denen verschiedene Funktionen verwendet werden. Um für die neuronale Regelung eine (oder mehrere) Aktivierungsfunktion(en) auswählen zu können, folgt zunächst die Erläuterung einiger Funktionen.

Die einfachste Form der Aktivierungsfunktion ist eine **binäre Schwellenwertfunktion**. Liegt der Eingangswert unterhalb des definierten Schwellenwertes θ, gibt die Funktion null aus, andernfalls eins (siehe auch Gleichung 2.20). Als Schwellenwert wird die Stelle bezeichnet, an der die Aktivierungsfunktion den größten Anstieg hat [27, S. 65]. Zur Veranschaulichung dient Abbildung 2.15, wo ein Schwellenwert von $\theta = 1$ gewählt wurde.

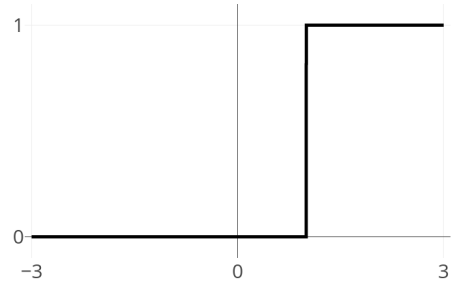

Abbildung 2.15: Graph einer Schwellenwertfunktion mit der Schwelle $\theta = 1$

$$f(x) = \begin{cases} 0 & x < \theta \\ 1 & x \geq \theta \end{cases} \tag{2.20}$$

Ausgehend von dem beschriebenen (vereinfachten) biologischen Modell ist es naheliegend, diese Aktivierungsfunktion zu verwenden. Dies zeigt beispielsweise die Veröffentlichung von McCulloch und Pitts aus dem Jahr 1943 [34]. Sie stellten ein Modell von Neuronen mit dieser Aktivierungsfunktion vor, weshalb diese Neuronen auch als *McCulloch-Pitts-Neuron* bezeichnet werden [30, S. 13]. Ein Nachteil dieser Funktion ist, dass sie an der Stelle des Schwellenwertes $x = \theta$ nicht ableitbar ist. Diese Eigenschaft ist notwendig, um das Lernverfahren der Fehlerrückführung (*Backpropagation* - 1974 von Paul Werbos eingeführt) anwenden zu können [27, S. 66].

Eine stetige Funktion mit einem ähnlichen Verlauf ist die **logistische Funktion**. Wie in der folgenden Abbildung (2.16) zu sehen ist, liegt der Wertebereich ebenfalls zwischen null und eins, wobei sich die Funktion diesen Grenzen nur asymptotisch nähert.

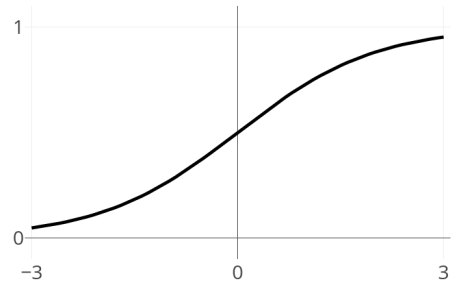

Abbildung 2.16: Verlauf der logistischen Funktion

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.21}$$

$$\sigma'(x) = x(1 - x) \tag{2.22}$$

Der größte Anstieg und damit der Schwellenwert befindet sich an der Stelle $\theta = 0$. Im Vergleich zu der Sprungfunktion erreicht die logistische Funktion ihr Maximum

erst im Unendlichen und nicht schon am Schwellenwert. Außerdem unterscheidet sich der zurückgegebene Wert kurz vor der Schwelle (z.B. $f(\theta - 0,1)$) und nach der Schwelle (z.B. $f(\theta + 0,1)$) nicht so stark. Dadurch bildet die logistische Funktion das Verhalten eines natürlichen Neurons noch besser ab als ein Schwellenwertelement. Da diese Funktion stetig und somit an jeder Stelle ableitbar ist, kann sie in Verbindung mit dem Backpropagation-Lernverfahren verwendet werden. Die Verwendung von nicht-linearen Aktivierungsfunktionen führt außerdem dazu, dass ein Neuronales Netz auch nicht-lineare Zusammenhänge abbilden kann. [33, S. 19]

Ein Nachteil dieser Funktion ist allerdings, dass besonders bei der Verwendung in tiefen Neuronalen Netzen das *vanishing gradient*-Problem auftreten kann. Für Anwendungsfälle mit null-zentrierten Daten, kann die Verwendung einer nicht null-zentrierten Aktivierungsfunktion außerdem zu ungewünschtem Verhalten während des Lernprozesses führen. [25]

Die **Tangens hyperbolicus** Funktion (*tanh*) ist wie auch die logistische Funktion eine Sigmoidfunktion. Der Wertebereich liegt allerdings zwischen minus eins und eins ($W = [-1, 1]$). Diesen Grenzen nähert sich die Funktion ebenfalls asymptotisch an.

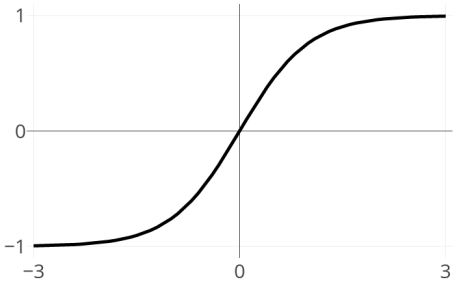

Abbildung 2.17: Verlauf der Tangens hyperbolicus-Funktion

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{2}{1 + e^{-2x}} - 1 \qquad (2.23)$$

$$tanh'(x) = 1 - tanh^2(x) \qquad (2.24)$$

LeCun et al. raten in *Efficient backprop* von der logistischen Funktion ab und empfehlen stattdessen *tanh*. Einer der Gründe ist eine in der Regel schnellere Konvergenz der Gewichte, d.h. der Fehler des Netzes verringert sich im Vergleich zu der logistischen Aktivierungsfunktion schneller (das Netz lernt schneller). [33]
Allerdings begünstigt auch diese Funktion das vanishing gradient-Problem, eignet sich aber für null-zentrierte Daten [25].

Eine weitere gebräuchliche Aktivierungsfunktion ist die **Rectified Linear Unit** (ReLU). Durch diese Funktion werden alle Eingabewerte kleiner als null auf null abgebildet. Ist der Eingabewert größer als null, wird er unverändert ausgegeben. Die folgende Gleichung (2.25) und Abbildung (2.18) veranschaulichen diese Aktivierungsfunktion.

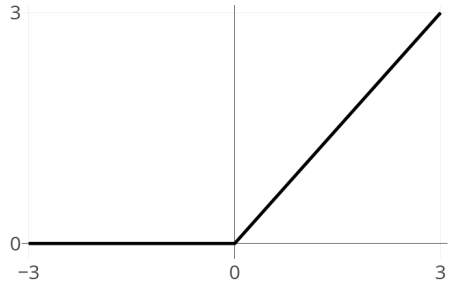

Abbildung 2.18: Ausgabe einer ReLU

$$f(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases} = max\,(0, x) \qquad (2.25)$$

Bei einer ReLU ist die Berechnung der Funktion und das Bilden der Ableitung deutlich weniger rechenaufwändig als bei den vorgestellten Sigmoidfunktionen. Aus dem Grund wird diese Funktion vor allem bei sehr großen Neuronalen Netzen eingesetzt, wie sie beispielsweise für die Bildverarbeitung notwendig sind. Außerdem konnte gezeigt werden, dass die Gewichte mit einer ReLU schneller konvergieren als mit den sigmoiden Funktionen [28]. Da es keine asymptotische Annäherung gibt, tritt auch kein *vanishing gradient* auf.
Da bei Anwendung dieser Funktion alle Werte unter null gleich null sind, kann es dazu kommen, dass Neuronen nicht mehr aktiviert werden (*dying ReLU*) [25].

Abhilfe schafft eine leichte Abwandlung, die sogenannte **Leaky ReLU**.

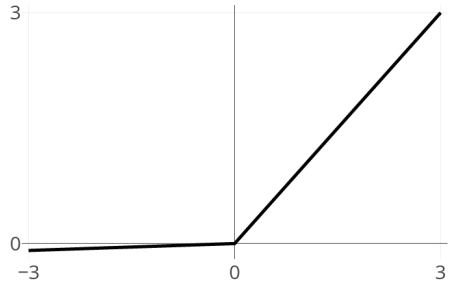

Abbildung 2.19: Ausgabe einer Leaky ReLU

$$f(x) = \begin{cases} 0.01x & x < 0 \\ x & x \geq 0 \end{cases} = max\,(0.01x, x) \qquad (2.26)$$

Bei dieser Variante gibt es eine leichte Steigung der Funktion im negativen Bereich, welcher ein *Absterben* von Neuronen verhindert (siehe Abbildung 2.19 und Gleichung 2.26). [25]

Welche der hier vorgestellten Aktivierungsfunktionen verwendet werden sollte, kommt hauptsächlich auf den Anwendungsfall und die zu verarbeitenden Daten an. Sowohl LeCun als auch Karpathy raten von der Verwendung der Logistikfunktion ab und empfehlen *tanh* bzw. wenn möglich die ReLU [33, 25].

2.2.6 Datenaufbereitung und Initialisierung

Die Wahl der Trainingsdaten hat Einfluss darauf, wie gut ein Netz lernt. Um den Trainingserfolg zu verbessern, sollten die Daten mithilfe der folgenden Verfahren an das zu trainierende Netz angepasst werden.

Ein Neuronales Netz konvergiert während des Trainings meist schneller, wenn der Durchschnitt über alle Trainingseingaben gleich null (**null-zentriert**) ist. Dazu wird für jedes Merkmal der Mittelwert über alle Trainingsdaten gebildet und von den einzelnen Datensätzen subtrahiert (vgl. mittlerer Graph in Abbildung 2.20). [33, 26]

Eine weitere Möglichkeit die Konvergenz mithilfe der Trainingsdaten zu verbessern, ist die **Normalisierung**. Die Jede Dimension der Daten wird dabei durch deren Standardabweichung dividiert, damit die Daten anschließend möglichst gleichverteilt sind (vgl. rechter Graph in Abbildung 2.20). Dies führt allerdings dazu, dass alle Trainingsdaten ungefähr den gleichen Einfluss auf den Lernprozess haben. In manchen Fällen kann dies nicht gewollt sein. [33, 26]

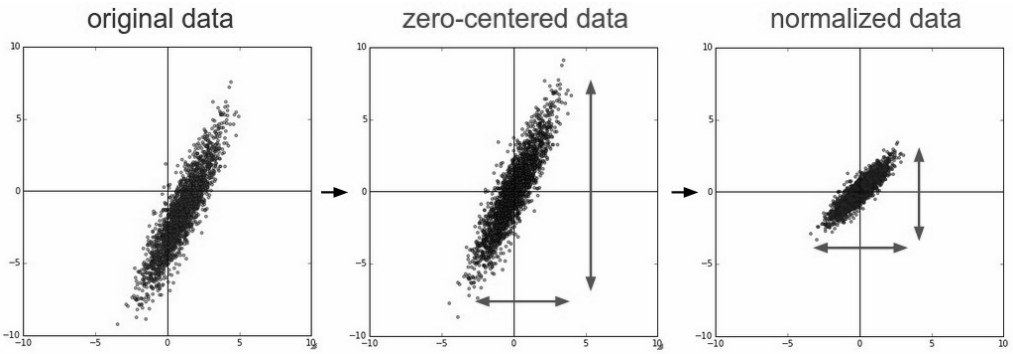

Abbildung 2.20: Schritte der Datenaufbereitung. [26]

Indem die Trainingsdaten in einem Koordinatensystem visualisiert werden, ist die Auswirkung der Null-Zentrierung sowie der Normalisierung zu erkennen (siehe Abbildung 2.20).

Ein weiterer Aspekt, welcher Einfluss auf den Lernvorgang hat, ist die **Initialisierung der Verbindungsgewichte**. Die Verbindungen sollten unterschiedliche Gewichte haben. Bei einem identischen Einfluss mehrerer Kanten auf den Fehler führt dies auch zu identischen Gewichtsänderungen. Im schlechtesten Fall erfolgen die Änderung dann immer synchron, wodurch das Netz allerdings weniger Informationen repräsentieren kann.
Außerdem konvergiert ein Netz schneller, wenn die Gewichte in dem Bereich des stärksten Anstiegs der Aktivierungsfunktion liegen. [33, S. 13] [27, S. 105]
Eine Faustregel besagt, dass für die *tanh*-Aktivierungsfunktion folgende Intervallgrenzen gewählt werden sollten:

$$Grenzwert = \pm\frac{1}{\sqrt{Verbindungen\ zu\ einem\ Knoten}} \tag{2.27}$$

2.2.7 Generalisierung

Eine wichtige Eigenschaft Neuronaler Netze ist die Fähigkeit zu generalisieren. So ist es bei der Bilderkennung gewünscht, dass ein Netz auch Bilder korrekt klassifiziert, die vorher nicht trainiert wurden.

Um das zu erreichen, ist die Vielfalt der Trainingsdaten ausschlaggebend. Wenn ein Netz beispielsweise Hunde auf Bildern erkennen soll und es mit Beispielen trainiert wurde, auf denen Hunde von vorn abgebildet sind, wird es auf einer Abbildung des Seitenprofils eines Hundes diesen nicht erkennen.

Neben der Wahl der Trainingsdaten gibt es noch Verfahren, welche die Generalisierungsfähigkeit eines Netzes verbessern.
Vor allem bei großen Netzen und zu langem Training kann es vorkommen, dass das Netz die Trainingsbeispiele auswendig lernt. Erkennbar ist das daran, dass Netzfehler mit den Trainingsdaten wesentlich niedriger sind als mit den Testdaten. Wenn der Lernvorgang früh genug abgebrochen wird (**early stopping**), kann das die Generalisierungsfähigkeit des Netzes verbessern. [27, S. 63]

Eine weitere Möglichkeit, wodurch hauptsächlich die Leistungsfähigkeit von tiefen Neuronalen Netzen verbessert werden kann, ist das **Dropout**-Verfahren. Während des Lernprozesses werden dabei zufällig Neuronen und deren Verbindungen ausgelassen. Dies betrifft sowohl die Berechnung der Ausgabe als auch das Anpassen der Verbindungsgewichte. [45]

2.3 Neuronale Netze in der Regelungstechnik

Wie in der Einleitung angedeutet, gibt es bereits Beispiele, wo Neuronale Netze für Regelungsaufgaben zum Einsatz kommen. Möglich ist dies, weil neuronale Regler ein beliebiges Optimierungsziel approximieren können. Ein Vorteil von Neuronalen Netzen gegenüber konventionellen, programmierten Regelsystemen ist die Lernfähigkeit, wodurch adaptive Regelungen möglich sind. Diese eignen sich vor allem für komplexe Regelsysteme, da Wissen über die inneren Vorgänge einer Strecke oder Regelung nicht unbedingt notwendig ist, um das Netz zu trainieren. Für das Training reicht es, wenn sich die Ausgabe des Netzes bewerten lässt. [35, S. 31+32]

Im Jahr 1990 erschien das Buch *Neural Networks for Control*, welches unter anderem eine Arbeit von A. Barto enthält [5]. In dem von ihm geschriebenen ersten Kapitel stellt er mehrere Methoden vor, wie Neuronale Netze Modelle von Systemen erlernen und diese Systeme regeln können. Auf diese Veröffentlichung beziehen sich weiterführende Arbeiten wie beispielsweise *Modellierung und Regelung mit Neuronalen Netzen* von Neumerkel und Lohnert [35]. Einige dieser Herangehensweisen für die Erstellung neuronaler Regelungen werden nun vorgestellt.

2.3.1 Kopieren einer Regelung

Die erste Variante ist das Kopieren einer vorhandenen Regelung, indem deren Eingangssignale (Sollwerte und Ausgabe der Stecke) auch als Eingangssignale eines Neuronalen Netzes verwendet werden. Nachdem die Regelgrößen bestimmt wurden, dienen diese Werte als Trainingsvorgabe für das Netz (siehe Abbildung 2.21).

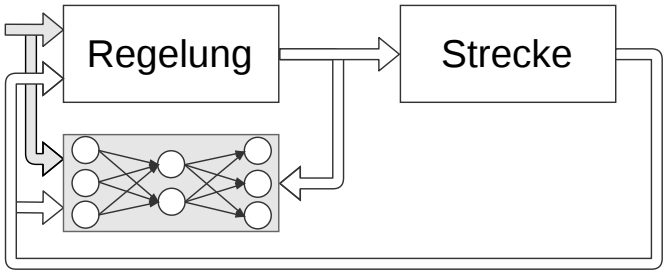

Abbildung 2.21: Kopie einer Regelung nach BARTO [5]. Die Pfeile mit der grauen Füllung (links) stellen die vorgegebenen Sollwerte dar.

Nach der Trainingsphase ist das Netz in der Lage, sich ähnlich wie die Regelung zu verhalten. Eine Verbesserung der Regelung kann dadurch aber nicht erreicht werden. Nach BARTO hat diese Methode allerdings eine Berechtigung, wenn der Einsatz der vorhandene Regelung Einschränkung/Schwierigkeiten mit sich bringt, die ein NN nicht hätte. Ein Beispiel dafür ist ein Arbeiter, der nur zu bestimmten Zeiten zur Verfügung steht. [5, S. 29f] [35, S. 33]

2.3.2 Adaptive Vorhersagen

Der in Abbildung 2.22 dargestellte Aufbau ermöglicht es, ein Netz für adaptive Vorhersagen zu trainieren. Zur Vereinfachung dienen die aktuellen Signale als Sollwerte für die Ausgabe. Die Eingänge des Netzes bekommen die aktuellen Signale verzögert, wodurch das Netz darauf trainiert werden kann, künftige Signale vorherzusagen. [5, S. 30]

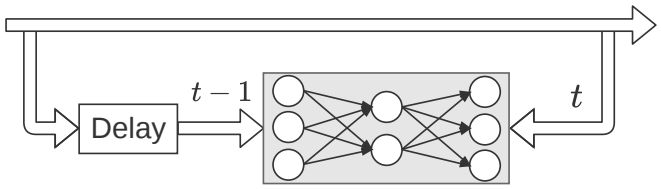

Abbildung 2.22: Neuronales Netz für adaptive Vorhersagen nach BARTO [5]

Die *Delay*-Einheit ist ein Zwischenspeicher, welcher die Eingangssignale um einen Zeitschritt verzögert $(t-1)$ ausgibt.

2.3.3 Bilden der Systeminversen

Ein weiterer Ansatz, wie mit einem Neuronalen Netz eine Regelung optimiert werden kann, ist das Bilden der Systeminversen. Dafür werden zum Training die Ausgänge

einer Regelstrecke an die Eingänge des Neuronalen Netzes angeschlossen und die Regelgrößen sind die Sollwerte (vgl. Abbildung 2.23).

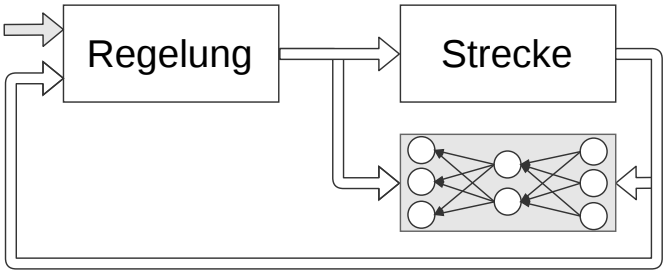

Abbildung 2.23: Erstellung eines inversen Streckenmodells nach BARTO [5]

Das so gebildete inverse Modell der Strecke wird anschließend vor die Strecke geschaltet, was zur Kompensation deren Nichtlinearität führt. Die Regelung der linearen Anteile kann dann beispielsweise ein PI-Regler übernehmen. Allerdings lässt sich nicht von jedem dynamischen System eine Inverse bilden. Der Erfolg dieser Methode hängt also von der Charakteristik des nachzubildenden Systems ab. [5, S. 34] [35, S. 33]

2.3.4 Direkte Regelung

Ebenso ist ein Regelkreis denkbar, in welchem ein Neuronales Netz eine Strecke direkt regelt. Die Ausgänge des Netzes werden dazu mit den Regelgrößen des zu regelnden Systems (Strecke) verbunden (vgl. Abbildung 2.24).

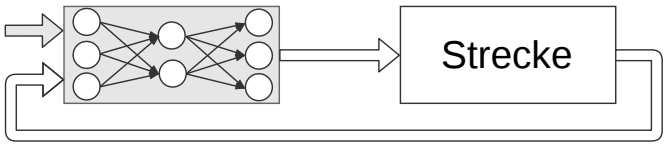

Abbildung 2.24: Direkte Regelung nach BARTO [5]

Trainiert werden kann das Netz vor dem Einsatz beispielsweise mit erfassten Trainingsdaten oder an einer Simulation. Erst das trainierte Netz wird dann mit der Strecke verbunden. Eine weitere Möglichkeit besteht darin, ein Neuronales Netz untrainiert mit dem zu regelnden System zu verbinden. Voraussetzung für das Training ist dabei, dass der gemachte Fehler der Strecke messbar ist und auf die Ausgabe des Netzes zurückgeführt werden kann. Laut NEUMERKEL und LOHNERT besteht allerdings auch die Möglichkeit, dass es keinen mathematischen Zusammenhang zwischen dem Sollverhalten der Strecke und den Parametern des Neuronalen Netzes gibt. In

einem solchen Fall wäre diese Methode ungeeignet. [35, S. 35]

Welche Herangehensweise, sich am besten für die Regelungsaufgabe des Betonverteilers eignet, wird im Folgenden diskutiert.

Kapitel 3

Ansatz

Basierend auf den Grundlagen des vorherigen Kapitels kann nun eine neuronale Regelung modelliert werden. In diesem Kapitel ist eine mögliche Herangehensweise für diesen Prozess beschrieben. Die entsprechenden Überlegungen für jeden Schritt in Bezug auf die Regelung des Betonverteilers sind im darauffolgenden Kapitel (4) dargelegt.

3.1 Schritt 1: Beschaffung von Trainingsdaten

Damit eine neuronale Regelung erstellt oder eine vorhandene Regelung um ein Neuronales Netz erweitert werden kann, ist der Zugang zu Trainingsdaten zwingend erforderlich.

Wenn möglich, können im laufenden Betrieb Daten der Anlage (Strecke) und der aktuellen Regelung gespeichert werden, um damit ein Netz zu trainieren oder eine Simulation zu erstellen. Relevant sind dafür die Eingangs- und Ausgabewerte der Regelung sowie der Strecke.
Wenn aus keinem bereits vorhandenen System Daten extrahiert werden können, es aber Expertenwissen über die Anlage gibt, besteht die Möglichkeit, eine Simulation zu entwerfen und daran die Regelung zu trainieren.
Ohne Expertenwissen bleibt die Option, eine neuronale Regelung direkt in Verbindung mit der Strecke zu trainieren (siehe 2.3.4 Direkte Regelung). Dabei ist zu beachten, dass zu Beginn des Trainingsprozesses häufig Fehlverhalten auftreten wird. Die Erstellung eines dafür geeigneten Trainingsmodells ist allerdings nicht Teil dieser Arbeit.

3.2 Schritt 2: Auswahl der Regelarchitektur

Nachdem die erforderlichen Trainingsdaten gesammelt wurden, geht es in diesem Abschnitt darum, eine der Problemstellung entsprechende Regelungsarchitektur zu erstellen.
In Abschnitt 2.3 wurde vorgestellt, wie laut A. Barto bzw. Neumerkel und Lohnert Neuronale Netze für Regelungsaufgaben verwendet werden können. Es zeichnen sich dabei zwei Tendenzen ab. Einerseits können Neuronale Netze dazu eingesetzt werden, um ein Modell der Strecke abzubilden und die Regelaufgabe für eine konventionelle Regelung (z.B. PI-Regler) zu vereinfachen [35, S. 33]. Ein Beispiel dafür ist das Bilden der Systeminversen (siehe Abschnitt 2.3). Die Alternative dazu stellt die direkte Verwendung eines Netzes als Regler dar (Abschnitt 2.3.4). [5] [11]

Eine Möglichkeit um eine individuelle Regelungsarchitektur zu erstellen, ist die Auswahl einer der vorgestellten Methoden und deren Anpassung an den Anwendungsfall. Die Tabelle 3.1 gibt eine situationsabhängige Empfehlung, welche der vorgestellten Regelungstechniken sich als Grundlage eignen.

vorhandene Trainingsdaten	**Art der Regelung**
Regelung & Strecke	direkte Regelung (trainiert/untrainiert), Kopie d. Regelung, Kopie d. Regelung + Reinforcement Learning, Modell der Strecke
nur Strecke	direkte Regelung (trainiert/untrainiert), Modell der Strecke
nur Regelung	direkte Regelung (untrainiert), Kopie d. Regelung, Kopie d. Regelung + Reinforcement Learning
keine	direkte Regelung (untrainiert)

Tabelle 3.1: Situationsabhängiger Vorschlag für die Auswahl einer Regelungstechnik

Wie in der Tabelle erkennbar, hat der Zugang zu Trainingsdaten einen großen Einfluss auf die Auswahl.

Im besten Fall ist das Training eines Netzes an einem vorhandenen System möglich. Das heißt, es können Trainingsdaten von einer bestehenden **Regelung und der**

Strecke bzw. von einer Simulation gewonnen werden. Daraus ergeben sich in Bezug auf die Architekturauswahl keine Einschränkungen.

Denkbar ist auch die Erstellung einer vollständig neuen Regelung, wofür es ein **Modell der Strecke** gibt, welches den Zusammenhang zwischen der Ein- und Ausgabe beschreibt.
Anhand des Streckenmodells könnte eine Simulation entwickelt und daran ein Neuronales Netz für die direkte Regelung trainiert werden. In Tabelle 3.1 steht *direkte Regelung (trainiert)* dafür, dass das Neuronale Netz vor dem Einsatz an der (realen) Strecke trainiert wurde. Das Netz zu trainieren, während es mit der Strecke verbunden ist (*direkte Regelung (untrainiert)*), hat den Vorteil, dass Fehler durch Abstrahierung vermieden werden, die bei der Entwicklung einer Simulation auftreten können. Weiterhin ist auch der Aufbau eines Neuronalen Modells der Strecke möglich, was z.B. eine Regelung mit PI-Reglern vereinfacht (*Modell der Stecke*) [35, S. 33].

Wenn es von der Strecke keine Trainingsdaten gibt und auch keine Simulation erstellt werden kann, bleibt die Möglichkeit einer *direkten, untrainierten Regelung*. Das Training findet dann durch Kopplung des Netzes an die Strecke und die Rückführung der gemachten Fehler auf das Neuronale Netz statt (siehe 2.3.4). Eine Alternative für diesen Fall gibt es, wenn **Trainingsdaten einer bestehenden Regelung** vorliegen oder sich diese mit einer Simulation erzeugen lassen. Dann kann ein Neuronales Netz die *Regelung kopieren* und wenn möglich durch Reinforcement Learning im laufenden Betrieb oder in einer Anlernphase weiter verbessert werden.

Wenn es für eine Strecke noch keine Regelung gibt, schließt dies das *Kopieren einer Regelung* aus. Das gilt ebenso, wenn der Zugang zu einer funktionierenden Regelung und das Domänenwissen für die Erstellung einer Regelungssimulation fehlt. Können zusätzlich keine Trainingsdaten einer vorhandenen Regelung verwendet werden, bleibt bei den vorgestellten Arten der Regelung nur die Möglichkeit, eine *untrainierte direkte Regelung* zu verwenden.

Nachdem entschieden wurde, in welcher Form ein Neuronales Netz in eine Regelung integriert wird, können die Eigenschaften des zur Problemstellung passenden Netzes bestimmt werden.

3.3 Schritt 3: Auswahl des Neuronalen Netzes

Nachdem die Trainingsdaten vorbereitet und der geeignete Aufbau der Regelung ausgewählt wurde, geht es in diesem Abschnitt darum, ein der Problemstellung entsprechendes Neuronales Netz zu wählen.
Die grundlegenden Arten Neuronaler Netze wurden bereits in Abschnitt 2.2.3 vorgestellt. Wie auch bei dem Aufbau der Regelung kann aber aufgrund der vielen Variationsmöglichkeiten hier nur eine grobe Orientierung gegeben werden.

3.3.1 Netzart

Abhängig von der Regelungsaufgabe kann zunächst entschieden werden, ob ein vorwärtsgerichtetes Netz (Multilayerperceptron) genügt oder die Regelung in der Lage sein muss, sequentielle Daten zu verarbeiten. Bei sequenziellen Daten ist der Einsatz eines rekurrenten Netzes empfehlenswert (siehe Multilayerperceptron).

3.3.2 Netzgröße

Die Größe der Ein- und Ausgabeschicht ergibt sich aus dem Kontext. Bei einer direkten Regelung hat die Eingabeschicht so viele Neuronen, wie es Soll- bzw. Vorgabewerte für den Regelkreis gibt. Die Anzahl der Ausgabeneuronen entspricht der Anzahl der Regelgrößen für die Strecke. Diese Hinweise gelten nur für Schichtenbasierte Netze. Die Anzahl der Neuronen eines Hopfield-Netzes wird beispielsweise nur durch die Größe des zu lernenden Musters bestimmt (siehe Hopfield-Netz).

Ein Neuronales Netz ohne versteckte Neuronen, wie beispielsweise das Singlelayerperceptron, ist in seiner Speicherkapazität und der Fähigkeit Funktionen zu approximieren eingeschränkt (siehe Singlelayerperceptron). Bei einem nicht-linearen Problem sollte darum ein Netz mit mindestens zwei Schichten verwendet werden. Wie Cybenko 1989 gezeigt hat, kann ein zweischichtiges Netz mit nicht-linearen Aktivierungsfunktionen jede beliebige stetige Funktion approximieren [7].
Unter Umständen kann ein dreischichtiges Netz die Problemstellung allerdings besser erlernen als ein zweischichtiges. Da mit jeder zusätzlichen Schicht auch weitere Nebenminima zu der Fehlerfunktion hinzukommen, empfiehlt es sich, zunächst mit einer versteckten Neuronenschicht zu beginnen und erst bei Bedarf die Anzahl zu erhöhen. [27, S. 103]

3.4 Schritt 4: Training des Neuronalen Netzes

Unabhängig von der Art der Regelung folgt als nächstes die Implementierung und das Training des zuvor ausgewählten Netzes. Welches Trainingsverfahren für welches Netz geeignet ist, wurde bereits im Kapitel 2 dargelegt. Zu beachten sind außerdem die Hinweise bezüglich der Initialisierung des Netzes und der Aufbereitung der Daten (Abschnitt 2.2.6). Es wird an dieser Stelle außerdem davon ausgegangen, dass die Netzart, die Trainingsdaten, das Trainingsverfahren und der Trainingszyklus (online-learning, batch-learning, ...) korrekt gewählt wurden.

Der Trainingsprozess eines Neuronalen Netzes enthält nach Lämmel und Cleve zudem auch weitere Anpassungen der Netzparameter, falls die bisher gewählte noch nicht die gewünschte Qualität erreicht. In der folgenden Abbildung ist dieser Trainingsprozess leicht verändert beschrieben (siehe Abbildung 3.1).

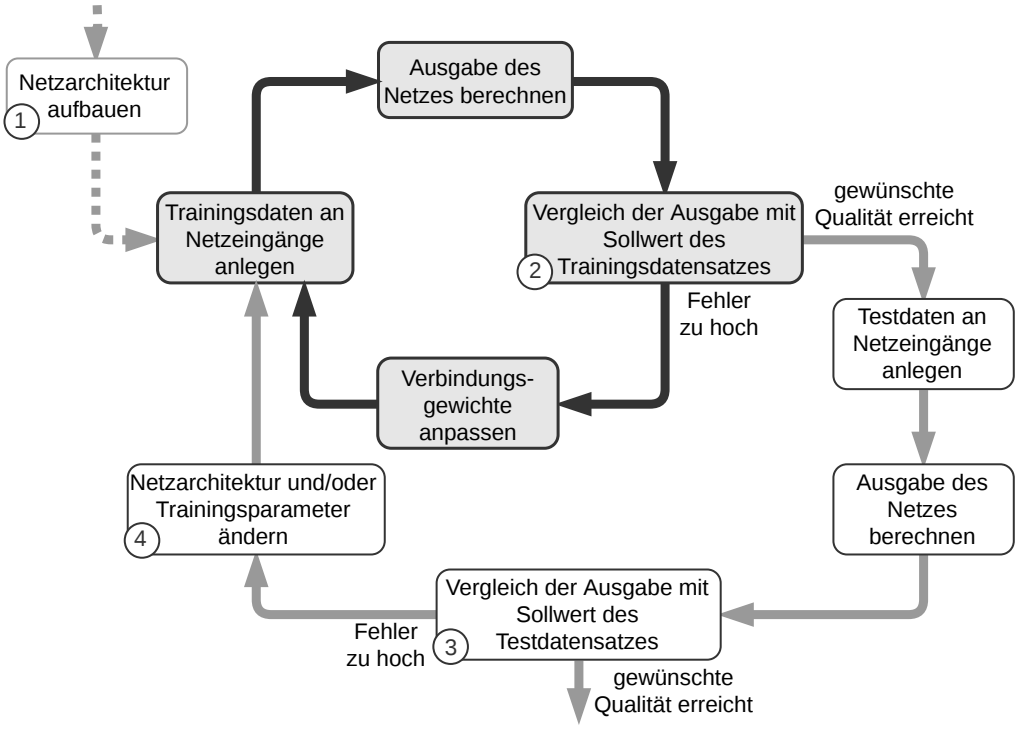

Abbildung 3.1: Trainieren eines Neuronalen Netzes (nach [31, S. 203])

Der Ablauf des Trainingsprozesses beginnt in Abbildung 3.1 links oben mit dem *Aufbau der Netzstruktur* (1). Das erstellte Netz wird zunächst mit den Trainingsdaten trainiert (zyklische Anpassung der Gewichte - schwarze Pfeile), bis ein Kriterium für

das Beenden des Trainings erfüllt wird (*Gewünschte Qualität erreicht* - 2). Es folgt die Prüfung der Netzausgabe mit dem Testdatensatz. Ist die Fehlerrate auch mit den Testdaten niedrig genug (3), kann zum nächsten Schritt übergegangen werden. [31, S. 203]

Ist das gewünschte Ergebnis noch nicht erreicht, kann eine *Änderung der Netzarchitektur und/oder der Trainingspatameter* (4) eine Verbesserung bewirken. Im Folgenden sind Einflussfaktoren auf das Trainingsergebnis und entsprechende Änderungsvorschläge zusammengefasst, deren Anwendung bei einem unbefriedigenden Ergebnis in Betracht gezogen werden kann.

Initialisierung der Verbindungsgewichte: Die Initialisierung der Verbindungsgewichte bestimmt den Ausgangspunkt für den Gradientenabstieg. Liegt dieser in der Nähe eines lokalen Minimums der Fehlerfunktion, kann dies zu schlechten Trainingsergebnissen führen. Eine erneute Initialisierung oder das Anwenden von Gleichung 2.27 behebt das Problem möglicherweise.

Lernrate: Wie in Abschnitt 2.2.4 erläutert, kann eine zu große Lernrate dazu führen, dass ein Minimum nicht gefunden wird. Eine zu kleine Lernrate bewirkt möglicherweise, dass der Trainingsalgorithmus ein lokales Minimum der Fehlerfunktion nicht mehr verlassen kann. Aus diesem Grund sollten Trainingsversuche mit verschiedenen Lernraten durchgeführt oder eine dynamische Lernrate implementiert werden.

Neuronenanzahl: Sind in einem Netz zu viele versteckte Neuronenschichten oder zu viele Neuronen in einer Schicht vorhanden, beeinflusst das den Lernerfolg (siehe Abschnitt 3.3.2). Das Trainieren und der anschließende Vergleich verschiedener Netzgrößen ist deswegen empfehlenswert.

Am Ende dieses vierten Schrittes sollte ein Netz vorliegen, dessen Ausgabe eine ausreichende Qualität hat, um als Teil eines Regelkreises in Betrieb genommen zu werden. Außerdem muss das Netz in der Lage sein, in Bezug auf seinen Einsatzbereich ausreichend generalisieren zu können.

3.5 Schritt 5: Test und Inbetriebnahme

Da dieser Schritt für jeden Anwendungsfall sehr individuell ist, können in diesem Abschnitt nur allgemeine Hinweise gegeben werden, wie eine Neuronale Regelung auf den Produktiveinsatz vorzubereiten ist.

Wenn für das Training des Neuronalen Netzes eine Simulation verwendet wurde, sollte diese so realitätsnah wie möglich modelliert sein. Bekommt das Netz beispielsweise Eingabewerte, die weit außerhalb der Trainingsdaten liegen, kann dies zu Fehlverhalten führen. Die Trainingsdaten des Netzes sollten extremer gewählt werden, als die Realität, um die Regelung möglichst robust gegen unvorhergesehene Situationen zu machen.
Unabhängig davon sollten zusätzlich Grenzen der Regelgrößen definiert werden.

Wenn die Möglichkeit besteht, hat es Vorteile, die neuronale Regelung parallel zu der konventionellen Regelung zu implementieren. Dadurch ist es möglich, die entwickelte Regelung zu testen aber im Zweifelsfall auf die konventionelle Regelung ausweichen zu können.

Kapitel 4

Umsetzung

Die im Kapitel *Ansatz* vorgestellte Herangehensweise wurde an dem Beispiel der Regelung des Betonverteilers angewendet. Alle damit verbundenen Überlegungen und deren Umsetzung sind Inhalt dieses Kapitels.
Bei der Umsetzung sind vor allem die ersten Schritte aus dem vorherigen Kapitel stark von der Aufgaben- bzw. Problemstellung abhängig. Deswegen wurde zu Beginn zunächst die Ausgangssituation analysiert.

Die Unitechnik-Systems GmbH betreut mehrere Betonfertigteilwerke, bei denen sich Betonverteiler im Produktiveinsatz befinden. Aufgrund unterschiedlicher Bedürfnisse der Kunden und Alter der Anlage unterscheiden sich die im Einsatz befindlichen Modelle (siehe Abschnitt 2.1.2). Da neue Betonverteiler hauptsächlich mit Förderschnecken und Klappen an den Auslassöffnungen ausgeliefert werden, ist die im Rahmen dieser Arbeit entwickelte Regelung, auf dieses Modell ausgerichtet.

Es ist zu untersuchen, ob eine Regelung mit Neuronalen Netzen bessere Ergebnisse erzeugen kann, als die bisher eingesetzte Speicherprogrammierbare Steuerung (SPS).

4.1 Erzeugung der Trainingsdaten

Die Extraktion von ausreichend Trainingsdaten aus dem Produktivsystem eines Betonwerks stellte sich in mehreren Gesprächen mit einem Experten als nicht praktikabel heraus. Eine Regelungseinheit, wie sie in aktuellen Anlagen zum Einsatz kommt (Siemens S7), war ebenfalls nicht für die Datengewinnung verfügbar. Daten aus dem Leitrechner (siehe beiliegender Datenträger) und einige Bildschirmaufnahmen von Regelkurven (wie z.B. in Abbildung 4.1) stehen allerdings zur Verfügung.

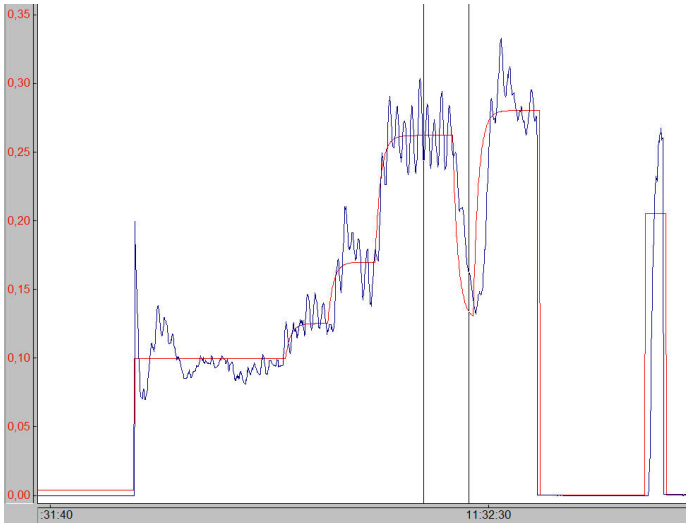

Abbildung 4.1: Gegenüberstellung von Sollwert- und Istwertkurve (soll: rot, ist: blau) eines realen Betoniervorgangs mit der konventionellen Regelung. Das dargestellte Intervall ist ca. 1 Minute und 15 Sekunden groß.

Da auch Expertenwissen vorhanden war, wurde entschieden, zunächst eine Simulation der Strecke aufzubauen. Zur Überprüfung des Streckenmodells wurde zusätzlich auch der aktuell verwendete Regelkreis in vereinfachter Form implementiert. Auf dieser Grundlage lässt sich ein Neuronales Netz trainieren und anschließend auch die neuronale Regelung mit dem Modell der konventionellen Regelung vergleichen. In den folgenden Abschnitten wird erläutert, wie bei der Erstellung dieses Modells vorgegangen wurde.

4.1.1 Auswahl der Simulationsumgebung

Bevor es im nächsten Abschnitt um die Modellierung der Simulation geht, wird in diesem Abschnitt die Wahl einer geeignete Simulationssoftware oder -Bibliothek erläutert.
Anhand der folgenden Kriterien wurde eine geeignete Simulationslösung bestimmt:

- 1 (sehr wichtig): Abbildbarkeit des aktuellen Regelkreises
- 1 (sehr wichtig): Schnittstellen zur Datenübertragung an ein Neuronales Netz
- 2 (relativ wichtig): Verbreitung und Langzeitkompatibilität
- 3 (wichtig): Verständlichkeit
- 4 (etwas wichtig): Kosten

- 4 (etwas wichtig): Unterstützte Betriebssysteme

Wie in dieser Auflistung zu erkennen, ist die Abbildbarkeit des Regelkreises (Regler und Stecke) *sehr wichtig.*
Außerdem *sehr wichtig* ist, dass aus der Simulation die erforderlichen Trainingsdaten für das Neuronale Netz extrahiert werden können. Falls das Netz nicht ebenfalls in der Simulationsumgebung entwickelt wird, sind dazu Schnittstellen notwendig.
Wird eines dieser sehr wichtigen Kriterien nicht erfüllt, führt das zum Ausschluss der entsprechenden Software.
Die Verbreitung und damit auch die Menge von Informationen, die über eine Lösung auffindbar sind, haben einen großen Einfluss, wie nachhaltig die erstellte Simulation ist bzw. wie lang diese verwendet und weiterentwickelt werden kann.
Für die Erstellung einer Simulation spielt die Verständlichkeit des Modells eine *wichtige* Rolle. Vor allem bei der punktuellen Zusammenarbeit mit einem Experten hilft eine gute Verständlichkeit, um den Einarbeitungsaufwand für diesen so gering wie möglich zu halten. Dies kann beispielsweise durch eine geeignete graphische Oberfläche realisiert sein.
Etwas wichtig sind die Kosten der Lösung. Sie soll im Rahmen dieser Arbeit getestet werden können. Die Kosten beeinflussen auch den Aufwand für eine Weiterverwendung und -entwicklung. Entstehen für die Weiterverwendung beispielsweise hohe Kosten, kann dies ein Hinderungsgrund für den Einsatz der Technologie sein.
Da bei dem betreuenden Unternehmen hauptsächlich Windows- und Linuxgeräte in Verwendung sind, wäre eine Kompatibilität mit diesen Systemen hilfreich.

Diese Kriterien wurden auf die folgenden Lösungen angewendet, um eine geeignete Simulationsumgebung zu wählen.

MATLAB/Simulink ist eine Erweiterung von MATLAB und ermöglicht unter anderem das Erstellen und Simulieren von digitalen Schaltungen über eine grafische Oberfläche.
Eine MATLAB- und Simulink-Lizenz für den kommerziellen Einsatz kostet 3000 Euro und die Studentenversion 69 Euro [51]. Es gibt zudem Schnittstellen, womit sich beispielsweise Modelle an SIMATIC STEP 7 und damit auf eine SPS übertragen lassen [50]. Neben TCP Verbindungen zu anderer Software, ist es auch möglich MATLAB-Code als Modul in z.B. JAVA-, Python- und C++ Anwendungen zu integrieren [52] [49]. MATLAB gibt es für Windows, Linux und macOS.

WinFACT/BORIS wird durch das Ingenieurbüro Dr. Kahlert entwickelt und verkauft. WinFACT ist laut der Webseite ein „Programmsystem [...], das einerseits Werkzeuge zur Analyse, Synthese und Simulation konventioneller Regelungssysteme zur Verfügung stellt, andererseits aber insbesondere auch Komponenten zur Behandlung von Fuzzy-Systemen und Neuronalen Netzen beinhaltet.“ [21].
WinFACT ist nur für Windows verfügbar und eine Lizenz für BORIS kostet mindestens 620 Euro. Für Schüler/Studenten gibt es eine eingeschränkte Version für 25 Euro [20]. Der Referenzliste zufolge hat die Software eine große Verbreitung unter Hochschulen, Berufsschulen und Unternehmen im deutschsprachigen Raum. Es sind zudem beispielsweise TCP- und UPD-basierte Schnittstellen zum Datenaustausch mit anderen Anwendungen und Geräten vorhanden [19].

Scilab/Xcos wurde durch MATLAB inspiriert und gilt als dessen Open Source Alternative. Es enthält auch eine mit Simulink vergleichbare grafische Oberfläche (Xcos) für die Erstellung von Regelkreisen [41]. Die neuste Version vom 15.02.2018 kann für Windows, Linux und macOS auf der Webseite von Scilab kostenlos heruntergeladen werden[1].
Über eine Programmierschnittstelle (*Application Programming Interface* - API) ist es möglich auch externe C/C++ Bibliotheken und Funktionen auszuführen. Die Schnittstelle kann außerdem von Drittanbietersoftware genutzt werden, um Scilab für Berechnungen zu verwenden. [42]

Eigenentwicklung Die bisher genannte Software ist auf die Erstellung und Bearbeitung von Blockschaltungen ausgelegt. Für die Entwicklung Neuronaler Netze wird häufig eine Programmiersprache mit entsprechenden Bibliotheken verwendet. Aus diesem Grund hat eine selbst entwickelte Simulation den Vorteil, dass in der gleichen Sprache auch das Neuronale Netz erstellt werden kann. Dies erleichtert den Datenaustausch zwischen Simulation und Neuronalem Netz und damit auch dessen Training. Andererseits fehlt bei einer Eigenentwicklung die Unterstützung der graphischen Programmierung, wie sie Simulink oder Xcos bietet. Der Entwicklungsaufwand lässt sich allerdings auch durch die Verwendung von Bibliotheken wie beispielsweise *SimPy*[2] für Python oder *SimJulia*[3] für Julia verringern.

[1] siehe `http://www.scilab.org/en/download/`

[2] siehe `https://simpy.readthedocs.io/en/latest/` [Abgerufen am 27.03.2018]

[3] siehe `https://simjuliajl.readthedocs.io/en/stable/welcome.html` [Abgerufen am 27.03.2018]

Zusammengefasst sind die Merkmale der beschriebenen Implementierungsmöglichkeiten in der folgenden Übersicht (Tabelle 4.1).

[46] [47]

		MATLAB/ Simulink	**WinFACT/ BORIS**	**Scilab/ Xcos**	**Eigen-entwicklung**
Regelkreis abbildbar		ja	ja	ja	ja
Schnitt-stellen	Netzwerk	TCP, UDP	TCP, UDP	TCP, UDP	geringe sprachen-abhängige Ein-schränkungen
	Export als	Java, C++, Python	C	C/C++	
	Import von	C/C++, .NET, Java, Python		C/C++, Java, Python	
Verbreitung und Langzeit-kompa-tibilität	SO Fragen	77.944 (2.095)	0 (0)	519 (21)	Abhängig von Sprache und Bibliothek
	YouTube Abonnenten	105360	119	3550	
	YT Videos	1159	54	51	
	Google Trends	75,5	<1	1,3	
	Letzte Aktu-alisierung	17.03.18	04.04.17	15.02.18	
Verständlichkeit		Modellerstellung mit visueller Programmierung möglich			i.d.R. ohne GUI
Preis in EUR		3000 (69)	620 (25)	0 (0)	0 (0)
Unterstützte Betriebssysteme		Windows, Linux, macOS	Windows	Windows, Linux, macOS	i.d.R. keine Ein-schränkung

Tabelle 4.1: Vergleich von Simulationslösungen für Regelsysteme. Die in Klammern stehenden Angaben bei den SO Fragen beziehen sich auf die im Tabellenkopf an zweiter Stelle genannte Software (Simulink, BORIS und Xcos). Der Preis in Klammern gilt für die Studentenversion. SO: Stackoverflow, YT: YouTube

Alle genannten Möglichkeiten zur Implementierung der Regelstrecken-Simulation sind in der Lage die Regelung abzubilden (siehe Tabelle 4.1) und erfüllen damit das wichtigste Kriterium.

Falls die Implementation des Neuronalen Netzes nicht in der gleichen Umgebung wie die Simulation erfolgt, gibt es bei allen Lösungen auch Schnittstellen, worüber der Datenaustausch mit einem Neuronalen Netz möglich ist.

Um die Verbreitung einer Software abzuschätzen, wurde die Anzahl der Stackoverflow-Fragen, Youtube-Abonnenten, Youtube-Videos und die Anzahl der Suchanfragen über Google Trends[4] analysiert. Die Zahlen von Google Trends sind Vergleichswerte und stehen nicht für eine bestimmte Anzahl von Suchanfragen. In der Tabelle wurde für jeden Suchbegriff (*matlab + simulink*, *winfact* und *scilab + xcos*; *boris* wurde nicht mit einbezogen, da der Begriff nicht eindeutig auf die Software verweist) der Durchschnitt über den Zeitraum von April 2017 bis April 2018 gebildet. Für die YouTube-Metriken wurden die offiziellen Kanäle der Anwendungshersteller untersucht (MATLAB/Simulink: MATLAB, WinFACT/BORIS: Ingenieurbüro Dr. Kahlert, Scilab/Xcos: Scilab). Daran ist die Tendenz erkennbar, dass MATLAB/-Simulink im Vergleich zu den beiden anderen Softwareprodukten am weitesten verbreitet ist.

Jedes der betrachteten Softwareprodukte wird weiterentwickelt und die letzte Änderung liegt maximal ca. ein Jahr zurück. Somit ist davon auszugehen, dass die Anwendungen auch in den kommenden Jahren weiter verwendet werden können.

Bei dem Kriterium „Verständlichkeit“ haben die ersten drei Lösungen einen Vorteil gegenüber der Eigenentwicklung. Diese bieten alle die Möglichkeit, eine Schaltung bzw. einen Regelkreis mithilfe von Visueller Programmierung zu erstellen. Dafür gibt es bereits vorgefertigte Bausteine wie PID-Regler und Blöcke für verschiedene Rechenoperationen, es lassen sich aber auch benutzerdefinierte Blöcke erstellen. Durch die Visuelle Programmierung können Schaltungen effizient entwickelt werden. Bei einer Eigenentwicklung mit einer Programmiersprache wie Python, Julia, C++ und Java sind diese Grundlagen nicht gegeben und bei Bedarf muss eine graphische Darstellung des Modells erst noch entwickelt werden. Vor allem bei der Zusammenarbeit mit einem Regelexperten ist anzunehmen, dass dieser ein Blockschaltbild schneller verstehen kann, als das gleiche Modell in Form von Programmcode.

[4] siehe `https://trends.google.de/trends/explore?q=matlab%20%2B%20simulink,winfact,scilab%20%2B%20xcos`

Der Preis für eine MATLAB/Simulink Lizenz ist allerdings vergleichsweise hoch. Bei Scilab/Xcos und der Eigenentwicklung entstehen bei der Anschaffung keine Kosten. Beachtet werden sollte aber auch der höhere Entwicklungsaufwand, wenn keine der ersten drei Simulationslösungen zum Einsatz kommt.

Bei der Kompatibilität mit verschiedenen Betriebssystemen unterscheidet sich WinFACT/BORIS von den restlichen Optionen. Es ist nur mit Windows kompatibel, wohingegen die übrigen Lösungen Windows, Linux und macOS unterstützen.

Letztendlich wurde MATLAB bzw. Simulink ausgewählt, um die Simulation des aktuell vorhandenen Regelkreises zu modellieren. Simulink ist den anderen Lösungen abgesehen vom Preis in allen Kategorien überlegen. Aufgrund seiner großen Verbreitung ist davon auszugehen, dass ein damit erstelltes Modell auch zu einem späteren Zeitpunkt noch verwendet und die hier vorgestellte Umsetzung auch auf spätere Projekte angewendet werden kann.
Um den Regelkreis des aktuellen Betonverteilers zu modellieren, waren Absprachen mit dem Entwickler dieser Regelung notwendig. Die grafische Oberfläche in Simulink konnte somit auch genutzt werden, um Erklärungen und Absprachen effizient zu gestalten.
Gegen eine Eigenentwicklung wurde sich hauptsächlich entschieden, da der Fokus dieser Arbeit nicht auf der Erstellung der Simulation liegen sollte, sondern auf den Möglichkeiten Neuronaler Netze. Für die Eigenentwicklung wäre aber eine Einarbeitung in regelungstechnische Details notwendig gewesen, was in Simulink, BORIS oder Xcos durch Verwendung von vorgegebenen Komponenten teilweise vermieden werden kann.

In dem folgenden Abschnitt ist beschrieben, wie das Simulationsmodell des Regelkreises in Simulink erstellt wurde.

4.1.2 Simulation des Regelkreises

Der im Einsatz befindliche Regelkreis wurde in Abschnitt 2.1.3 grob beschrieben. Im Folgenden geht es um die Modellierung dieses Regelkreises in Simulink, wobei es insbesondere um den Aufbau des Regelbausteins geht. Der Teil des Modells, welcher den Betonverteiler simuliert, ist Inhalt des nächsten Abschnitts.

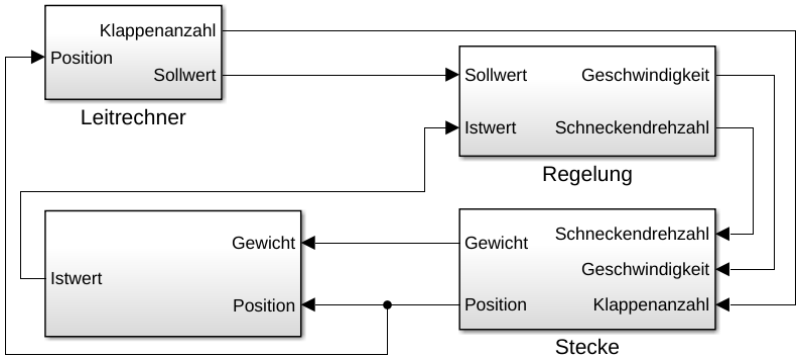

Abbildung 4.2: Das Blockschaltbild des konventionellen Regelkreises in Simulink.

Anhand von 4.2 wird das Simulationsmodell nun erläutert.
In dem Block mit der Bezeichnung *Leitrechner* befinden Sich zwei *1-D Lookup Table*-Blöcke. Diese greifen auf Daten zurück, welche nach dem Laden des Modells aus einer Excel-Tabelle importiert werden. Als Datenquelle diente zunächst der Auszug einer realen Sollkurve, welcher in Abschnitt 2.1.3 vorgestellt wurde. Für das Training des Neuronalen Netzes erfolgte die Implementierung eines Algorithmus, welcher zufällige Sollkurven erzeugt.
Anhand der übergebenen Position des Betonverteilers, geben die *Lookup Table*-Blöcke die zur Position gehörige Anzahl der offenen Klappen und den Sollwert aus.

Der Sollwert wird anschließend an den *Regelungs*-Block übergeben, dessen Schaltung in Abbildung 4.3 zu sehen ist.

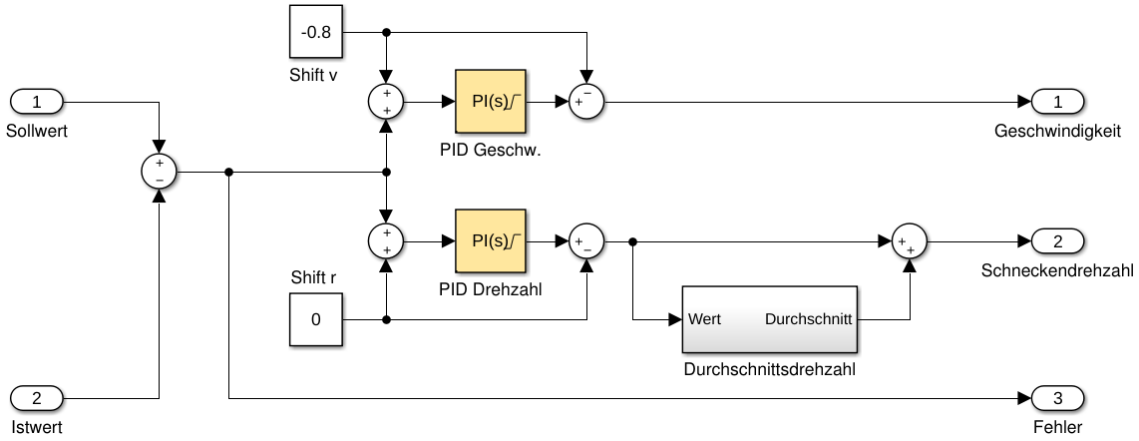

Abbildung 4.3: Innere Struktur des *Regelungs*-Blocks

Die PID-Regler-Blöcke in Simulink haben einen Eingang, an welchen der zu minimierende Wert s angelegt werden muss. Dieser Wert ergibt sich in diesem Fall

aus der Differenz von Soll- und Istwert. Der PID-Regler errechnet daraus nach der folgenden Gleichung eine *Sprungantwort* (Änderung der Regelgröße).

$$PID(s) = P + I\frac{1}{s} + D\frac{N}{1 + N\frac{1}{s}} \tag{4.1}$$

Wobei N ein frei wählbarer Filterkoeffizient ist. Bei den gelb dargestellten PID-Reglern wurde kein Wert für den D-Anteil gesetzt (bzw. $D = 0$), da dies auch in der realen Regelung nicht der Fall ist. Somit kann dieser Regler auch als PI-Regler bezeichnet werden und die obige Gleichung vereinfacht sich entsprechend. Die Regelparameter können beispielsweise durch das Verfahren von Ziegler und Nichols eingestellt werden [58]. Für das erstellte Modell eignen sich folgende Werte:

Regelgröße	**P-Anteil**	**I-Anteil**
Geschwindigkeit	-1,2	0
Schneckendrehzahl	0,3	1

Tabelle 4.2: Regelparameter des Modells in Simulink

In der SPS des Betonverteilers gibt es außerdem einen Parameter, der den Eingangswert des PI-Reglers anheben und den Ausgangswert absenken kann oder umgekehrt. In Abbildung 4.3 wurde dies durch *Shift v* und *Shift r* umgesetzt.

Weiterhin gibt es den benutzerdefinierten Block mit der Bezeichnung *Durchschnittsdrehzahl*, welcher den Durchschnitt über die Drehzahlen der letzten fünf Schritte bildet. Zum aktuellen Regelwert wird dieser Durchschnitt anschließend addiert.

Um den Fehler während des Betoniervorgangs ermitteln zu können, ist der dritte Ausgabewert des Regelungs-Blocks vorhanden, wobei Geschwindigkeit und Schneckendrehzahl die eigentlichen Regelgrößen sind.
Zusammen mit der Information, welche Klappen geöffnet sind, beeinflussen diese Regelgrößen das Verhalten des Betonverteilers. Die Simulation, welche den Betonaustrag und die Bewegung berechnet, ist aufgrund der Komplexität im folgenden Kapitel separat beschrieben.

Die Streckensimulation liefert das aktuelle Gewicht des Verteilers und dessen Position, welche die Eingangswerte für den Block links unten in Abbildung 4.2 sind.

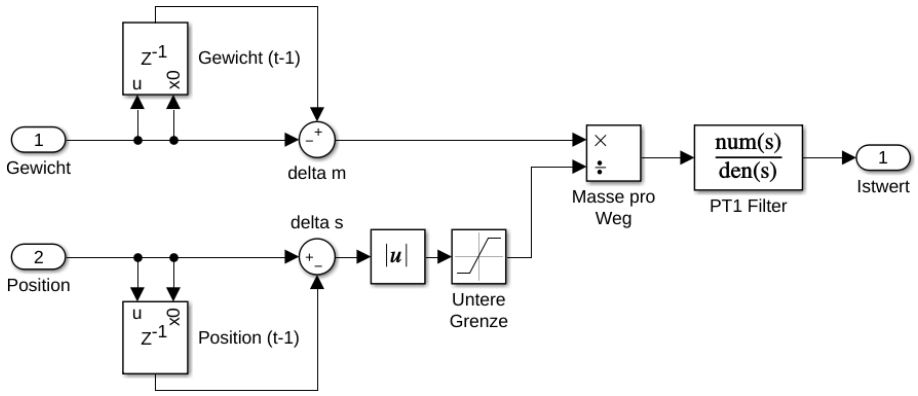

Abbildung 4.4: Schaltung zur Berechnung des Istwertes.

Gemäß Abbildung 4.4 wird als erstes der Gewichts- und Positionsunterschied (*delta m* und *delta s*) im Vergleich zum vorherigen Regelschritt gebildet. Die Blöcke *Gewicht (t-1)* und *Position (t-1)* enthalten die vorherigen Werte und werden im ersten Simulationsschritt mit dem aktuell anliegenden Wert über die Schnittstelle $x0$ initialisiert. Die Massedifferenz (*delta m*) wird anschließend durch den absoluten Wert des zurückgelegten Weges (|*delta s*|) dividiert. Um eine Division durch Null zu vermeiden, hat *delta s* einen sehr kleinen positiven Wert als untere Grenze.
Der so berechnete Istwert wird daraufhin noch in einem *PT1 Filter* durch die Gleichung

$$f(s) = \frac{1}{5s + 1} \tag{4.2}$$

gefiltert und dient im kommenden Simulationsschritt als Eingangswert für die Regelung. Die Koeffizienten dieser Gleichung wurden durch Ausprobieren ermittelt, sind aber frei wählbar.

4.1.3 Simulation des Betonverteilers

Die Simulation des Betonverteilers ist besonders wichtig, weil das Neuronale Netz bei fast allen vorgestellten Regelungsarchitekturen (Abschnitt 2.3) daran trainiert wird. Die einzige Ausnahme ist das Kopieren der Regelung, wo die vorhandene Regelung die Trainingsdaten liefert. Von der Genauigkeit dieser Simulation hängt somit ab, wie gut ein damit trainiertes Netz für die Regelung in einer realen Anlage eingesetzt werden kann. Um das Verhalten des Betonverteilers möglichst realitätsnah zu modellieren, wurden folgende Eigenschaften und Zusammenhänge identifiziert.

Eigenschaft (modelliert?)	**Beschreibung/Auswirkung**
Fließfähigkeit **(ja)**	Beschreibt wie flüssig oder fest der Beton ist und wird durch das Ausbreitmaß angegeben (normal: 35-50 cm)
Druck auf die Öffnungen	Umso mehr Beton in dem Kübel vorhanden ist, desto stärker wird der Beton aus den Auslassöffnungen gedrückt. (Förderrate steigt)
Trockendauer	Je mehr Zeit seit dem Mischen vergangen ist, umso mehr bindet der Beton ab (wird fester)
Klumpengröße und -anzahl **(ja)**	Mit abnehmender Fließfähigkeit steigt die Anzahl und Größe der Klumpen, die aus dem Betonverteiler kommen
Schwankungen der Fließfähigkeit	Enthält der Kübel beim Befüllen noch Restbeton, hat dieser meist eine geringere Fließfähigkeit als der Neue. Nachdem der Restbeton ausgetragen wurde, kommt es zu einem plötzlichen Anstieg der Fließfähigkeit

Tabelle 4.3: Eigenschaften und Zusammenhänge des Betons

Diese Tabelle (4.3) beinhaltet Eigenschaften des Betons. Modelliert wurde von diesen Aspekten die Fließfähigkeit in Zusammenhang mit der Klumpengröße und -anzahl, da es bei festem Beton zu einer stärkeren Klumpenbildung kommt als bei flüssigem. Die Schwankungen der Fließfähigkeit sind für eine Regelung besonders herausfordernd und sollten deswegen für das Training eines Neuronalen Netzes berücksichtigt werden, bevor es in der Realität eingesetzt wird.
Die Betoneigenschaften wirken sich außerdem auf die Eigenschaften der Anlage aus, die in der folgenden Tabelle aufgeführt sind.

Eigenschaft (modelliert?)	**Beschreibung/Auswirkung**
Anzahl der Schnecken/ Klappen **(ja)**	Je mehr Klappen geöffnet/Schnecken aktiv sind, desto mehr Beton kann ausgetragen werden
Drehzahl der Schnecken **(ja)**	Je höher die Schneckendrehzahl, desto mehr Beton wird pro Strecke ausgetragen (Istwert steigt)
Ansprechverhalten der Schnecken	Es dauert eine gewisse Zeit, bis eine Drehzahländerung umgesetzt wird
Förderrate einer Schnecke **(ja)**	Gibt an, wie viel Masse pro Umdrehung befördert werden kann
Maximaldrehzahl einer Schnecke **(ja)**	Technisch bedingte Höchstdrehzahl
Geschwindigkeit des Verteilers **(ja)**	Je höher die Geschwindigkeit, desto weniger Beton wird pro Strecke ausgetragen (Istwert sinkt)
Trägheit des Betonverteilers	Aufgrund der Massenträgheit gibt es eine Beschleunigungsphase, bis die gewünschte Geschwindigkeit erreicht ist
Maximalgeschwindigkeit des Kübels **(ja)**	Technisch bedingte Höchstgeschwindigkeit
Messstörungen **(ja)**	Die Gewichtsmesszellen und Positionsmessgeräte haben Messungenauigkeiten, welche mit der Fahrgeschwindigkeit steigen

Tabelle 4.4: Eigenschaften und Zusammenhänge des Betonverteilers

Die Förderrate wird beispielsweise wesentlich durch die Fläche des Schneckengewindes bestimmt. Die Fließfähigkeit des Betons und der Druck in dem Behälter haben aber ebenfalls einen Einfluss. So wird bei hohem Druck und hoher Fließfähigkeit mehr Beton pro Umdrehung ausgetragen, als wenn der Behälter fast leer ist.

Um zu zeigen, dass Neuronale Netze für Regelaufgaben geeignet sind, werden für die Simulation zunächst die in den Tabellen markierten Aspekte modelliert. Zusammengefasst beinhaltet das:

- Schneckendrehzahl und deren Obergrenze
- Förderrate der Schnecken
- Anzahl der geöffneten Klappen/der aktiven Förderschnecken
- Geschwindigkeit des Betonverteilers und deren Obergrenze
- Bildung von Betonklumpen (Fließfähigkeit)
- Messstörungen

Wenn ein Neuronales Netz nicht in der Lage ist, diesen vereinfachten Betonverteiler zu regeln, kann es für den Produktiveinsatz bereits ausgeschlossen werden. Eine Erweiterung des Simulationsmodells kann durchgeführt werden, sobald ein Netz gefunden wurde, welches in der Lage ist, dieses vereinfachte Modell zu regeln.

Das in Simulink erstellte Schaltbild für den Betonverteiler wird anhand der folgenden Abbildung erläutert.

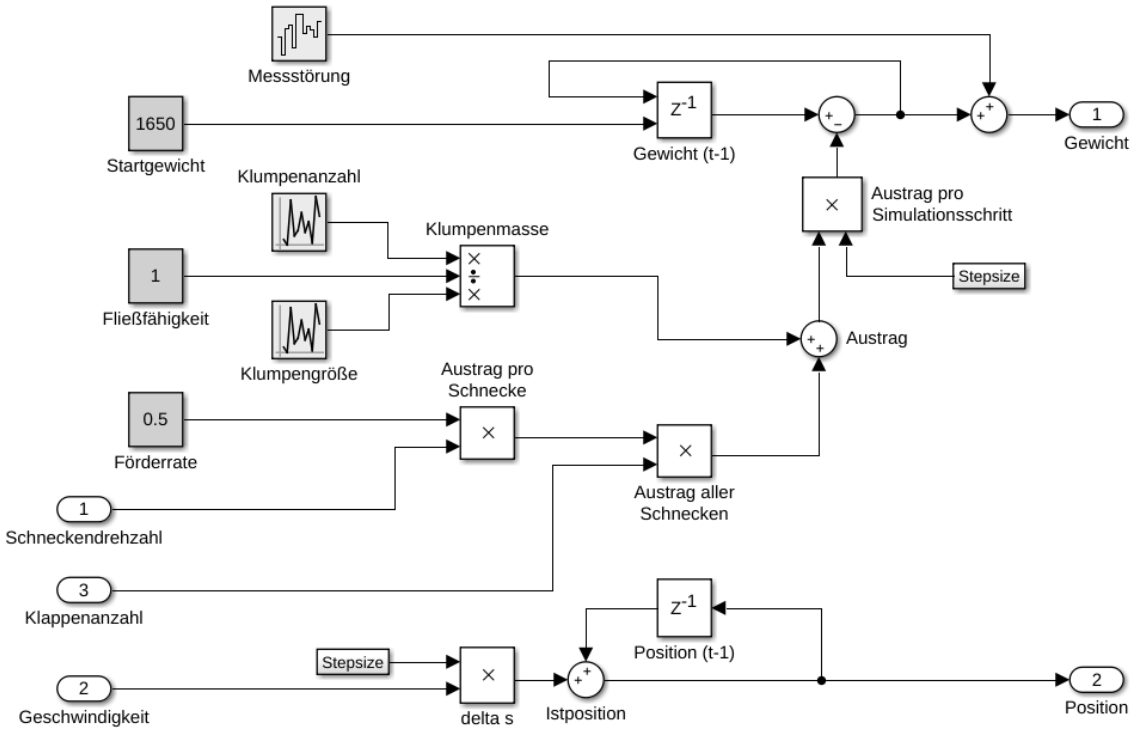

Abbildung 4.5: Schaltung zur Simulation von Gewicht und Position eines Betonverteilers.

Auf der linken Seite der Abbildung 4.5 befinden sich die oben genannten Aspekte (*Messstörung, Klumpenanzahl und -größe, ...*). Auf der rechten Seite ist jeweils ein Ausgang für das Gewicht und die Position des Betonverteilers. Das Startgewicht (dunkelgrauer Block) wird für jede Simulation neu initialisiert. In jedem Simulationsschritt wird zunächst der Austrag nach der Gleichung errechnet, welche auch aus der Abbildung ersichtlich ist:

$$Austrag = \Delta t \cdot (\frac{Klumpenanzahl \cdot Klumpengröße}{Fließfähigkeit} + Schneckendrehzahl \cdot Förderrate \cdot Klappenanzahl) \tag{4.3}$$

Die *Klumpenanzahl* und *-größe* (in kg) sind gleichverteilte, positive Zufallszahlen, welche sich indirekt proportional zur Fließfähigkeit verhalten. Anstelle einer kon-

stanten Fließfähigkeit wäre es realistischer, eine über die Zeit abnehmende Fließfähigkeit mit einem anschließenden plötzlichen Anstieg (siehe Tabelle 4.3) zu modellieren. Nicht berücksichtigt wurde außerdem der Einfluss der Fließfähigkeit auf die Förderrate.
Der Austrag wird anschließend von dem Gewicht aus dem letzten Simulationsschritt $(t-1)$ abgezogen und das Ergebnis in dem Baustein *Gewicht (t-1)* zwischengespeichert. Für das simulierte Gewicht des Betonverteilers gilt:

$$Gewicht(t) = Gewicht(t-1) - Austrag + Messstörung \tag{4.4}$$

Die Messstörung ist ein zufälliger, normalverteilter Wert mit dem Erwartungswert $\mu = 0$ und der Varianz $\sigma^2 = 0,002$.
Der Abstand zwischen den Simulationsschritten Δt (*Stepsize*) wird bei der Berechnung des *Austrags* und der *Position* mit einbezogen, da die Ausganswerte *Schneckendrehzahl, Klumpenanzahl* und *Geschwindigkeit* jeweils zeitabhängig sind.
Die aktuelle Position ist die Summe aus zurückgelegtem Weg Δs und der vorherigen Position:

$$\begin{aligned} Position(t) &= Position(t-1) + Geschwindigkeit \cdot \Delta t \\ &= Position(t-1) + \Delta s \end{aligned} \tag{4.5}$$

Durch die Simulation des Regelkreises ist es möglich, Trainingsdaten für ein Neuronales Netz zur Verfügung zu stellen, welches sich anschließend in eine Regelungsarchitektur einbinden lässt. Im nächsten Abschnitt wird die Wahl dieser Architektur für den konkreten Anwendungsfall erläutert.

4.2 Auswahl der Regelarchitektur

Dem vorgestelltem Ansatz aus Kapitel 3 zufolge, kann nach der Beschaffung der Trainingsdaten, die Wahl und Implementierung einer geeigneten Regelarchitektur erfolgen. In Betracht gezogen wurden die in Abschnitt 2.3 vorgestellten Architekturen.

Mit der erstellten Simulation lassen sich sowohl Trainingsdaten für die Regelung als auch für die Strecke erzeugen, wodurch sich zunächst keine Einschränkung in Bezug auf die Art der Regelung ergeben (vgl. 3.1).

Als erstes konnte das *Kopieren der vorhandenen Regelung* ausgeschlossen werden, weil dies nicht die gewünschte Optimierung bringt. Allerdings besteht die Möglichkeit, eine kopierte Regelung durch *Reinforcement Learning* weiter zu trainieren. Dazu muss das Ergebnis eines Regelvorgangs bewertet werden können. Eine Methode, die Austragsmenge automatisch zu bestimmen, wurde von K. HEIENBROK untersucht, aber wegen mangelnder Genauigkeit nicht in Produktivsysteme integriert [15, S. 29].
Denkbar wäre auch die Abweichung zwischen Soll- und Istwert als Fehlerfunktion zu definieren und nach jedem Betoniervorgang das Neuronale Netz mit Backpropagation zu trainieren. Wie gut das in einer SPS umsetzbar ist fraglich, wurde aber im Rahmen dieser Arbeit nicht untersucht.

Bei Erstellung und Einsatz eines *Streckenmodell*s sind Anpassungen der vorhandenen, konventionellen Regelung notwendig, welche Kenntnisse im Bereich der Regelungstechnik erfordern. Bei einer *direkten Regelung* ist dies nicht in dem Maß der Fall, weil die Regelungslogik vollständig innerhalb des Netzes liegt. Somit wurde die Architektur für eine direkte Regelung des Betonverteilers durch ein Neuronales Netz gewählt.

4.3 Das Neuronale Netz

Um die Eignung Neuronaler Netze für die Regelung eines Betonverteilers zu untersuchen, wurden mehrere Prototypen entwickelt und schrittweise erweitert. Dieses Kapitel dient dazu, die Wahl der Entwicklungsplattform und den Entwicklungsprozess der Prototypen zu erläutern.

4.3.1 Auswahl der Entwicklungsplattform

Wie auch bei der Wahl der Simulationslösung gab es seitens des betreuenden Unternehmens keine Einschränkung bei der Wahl einer geeigneten Programmiersprache, einer Bibliothek oder deren Kompatibilität mit einem bestimmten Betriebssystem.

J. F. PUGET zufolge, wird bei Stellenangeboten die Sprache Python am häufigsten in Verbindung mit „Machine Learning“ gebracht. An zweiter Stelle steht die Sprache R, anschließend kommen Java und C++. Er zeigt auch, dass der Einfluss der Sprachen Scala und Julia noch nicht groß, aber stark gestiegen ist. Diese Statistik

verdeutlicht aber auch, dass es mehrere Sprachen gibt, die in diesem Kontext genutzt werden. [37]
Die Dominanz von Python in Verbindung mit maschinellem Lernen und Neuronalen Netzen wird auch durch andere Quellen, wie den Fragen auf Stackoverflow deutlich. Dort wird das Tag *machine-learning* am häufigsten in Verbindung mit *python* (ca. 32%) verwendet[5]. Auch das Tag *neural-network* wird nach *machine-learning* am häufigsten mit *python* verbunden (ca. 27%)[6]. Entsprechend viele Beispiele und Bibliotheken (Tensorflow, scikit-learn, Theano, Caffe, ...) zu maschinellem Lernen gibt es für Python.
Eine Alternative ist beispielsweise Julia. Bei der Entwicklung dieser Programmiersprache wurde besonders auf Parallelität und Leistungsfähigkeit und Flexibilität Wert gelegt. So können beispielsweise C- sowie Fortran-Funktionen direkt sowie Python-Module über ein Paket eingebunden und verwendet werden. Außerdem erreichen Julia-Anwendungen durch den verwendeten *just-in-time* Compiler ähnliche Laufzeiten wie entsprechende C-Anwendungen. [6] [3]
Auch mit Matlab und Simulink lassen sich Neuronale Netze implementieren. Da der Regelkreis damit schon modelliert wurde, bietet sich diese Software auch für die Verwendung bei der Entwicklung Neuronaler Netze an. So könnte auf die Trainingsdaten für das Netz direkt zugegriffen werden. Der Nachteil an Simulink sind die hohen Kosten der dafür benötigten Zusatzpakete, wie der Neural Network Toolbox[7].

Anhand der folgenden Kriterien werden die drei in Betracht gezogenen Möglichkeiten (Python, Julia und MATLAB) verglichen.

- 1 (sehr wichtig): Abbildbarkeit eines Neuronalen Netzes
- 2 (relativ wichtig): machine-learning Bibliotheken mit Unterstützung für NN
- 2 (relativ wichtig): Bedienerfreundlichkeit (Debugging, graphische Ausgabe)
- 3 (wichtig): Verbreitung und Langzeitkompatibilität
- 3 (wichtig): Schnittstellen zu dem Modell in Simulink (TCP/UDP, ...)
- 3 (wichtig): Kosten

[5] siehe `https://stackoverflow.com/questions/tagged/machine-learning` [abgerufen: 02.04.18]
[6] siehe `https://stackoverflow.com/questions/tagged/neural-network` [abgerufen: 02.04.18]
[7] siehe `https://de.mathworks.com/store/link/products/standard/SL` [abgerufen: 03.04.18]

Die wichtigste Eigenschaft ist, dass es mit der Entwicklungsplattform möglich ist, ein Neuronales Netz zu implementieren und zu trainieren. Da dies bei allen drei Lösungen gegeben ist, wird dieser Aspekt im Folgenden nicht separat erwähnt. Damit ein Entwickler ein NN und dessen Trainingsalgorithmus nicht von Grund auf neu entwickeln muss, ist die Unterstützung von Frameworks wie *Tensorflow*, *Caffe*, *Theano* o.Ä. wünschenswert. Damit mögliche Fehler schnell identifiziert werden können, sollte außerdem eine Entwicklungsumgebung mit entsprechender Unterstützung für *Debugging* vorhanden sein. Analog zur Auswahl der Simulationsumgebung wird auch die Verbreitung und Langzeitkompatibilität sowie die Kosten in die Entscheidung mit einbezogen.

MATLAB/Simulink wurde aufgrund der Zusatzkosten und der fehlenden Unterstützung verbreiteter Bibliotheken ausgeschlossen. Damit blieb die Auswahl zwischen Python und Julia.
Um eine Entscheidung zu treffen, wurden Beispielimplementierungen von Neuronalen Netzen mit Julia und Python erstellt. Mit Entwicklungsumgebung *Juno*[8] für Julia ist es möglich, Programmcode zeilen- oder blockweise auszuführen, womit sich Teile einer Anwendung während der Entwicklung testen lassen. IPython-Notebooks[9] bieten ähnliche Funktionen und eigenen sich zu Demonstrationszwecken, allerdings weniger für die Entwicklung. Aufgrund der zu erwartenden Leistungsvorteile, die Unterstützung etablierter Bibliotheken für Maschinelles Lernen (z.B. Tensorflow und MXNet) sowie den interaktiven Entwicklungsprozess wurde Julia für die Implementierung des NN ausgewählt.

In Julia wurden daraufhin mehrere Module entwickelt, welche die Grundstruktur des Projektes bilden und im Folgenden erläutert werden.

4.3.2 Grundstruktur des Projekts

Der Quellcode zu diesem Projekt befindet sich auf dem Datenträger zu dieser Arbeit.

Das `src`-Verzeichnis enthält einen Unterordner `models` mit den beiden erstellten Prototypen, wobei der erste als `Jordan` und der zweite als `Custom` bezeichnet wird. In Anlehnung an die Komponenten des Regelkreises wurden außerdem die Module

[8] siehe `http://junolab.org` [abgerufen: 02.04.18]
[9] siehe `https://ipython.org` [abgerufen: 02.04.18]

`Master` und `Simulation` erstellt.
Im Folgenden werden die einzelnen Komponenten kurz beschrieben.

`Master` beinhaltet die vom Leitrechner zur Verfügung gestellte Funktionalität. Dazu gehört die Erzeugung einzelner Trainingssequenzen oder Trainingsdatensätze, welche mehrere Sequenzen enthalten. Außerdem stellt dieses Modul zwei Funktionen für die Konvertierung der Ein- und Ausgabewerte zur Verfügung.

`Simulation` ist das Modul, womit Parameter für die Simulation wie die Größe des Trainingsdatensatzes verwaltet werden. Außerdem enthält `Simulation` ein Julia-`Struct` mit der Bezeichnung `Dispenser`, welches in Verbindung mit der Funktion `calc_output` das Verhalten des Betonverteilers simuliert. An der Stelle wird nicht wie zuerst vorgesehen, auf das bereits vorhandene Simulink-Modell zugegriffen, da es sich als einfacher erwies, das Modell direkt in Julia zu implementieren.

`Jordan/Custom` enthält die für das Neuronale Netz benötigte Funktionalität. Dazu gehört ein Modell des Neuronales Netzes, Funktionen, um dessen Ausgabe zu berechnen sowie die passende Trainingsmethode.

`Benchmark` ist ein Julia-`Struct`, welches dazu dient, die für einen Test relevanten Konfigurationsparameter zu kapseln. Dies umfasst die Anzahl der durchzuführenden Versuche (`count`), eine Liste der Neuronenzahlen pro Schicht (`dims`) und die Lernrate (α).
Die Funktionen `run_benchmark` und `run_benchmarks` erwarten eine einzelne bzw. eine Liste solcher Konfigurationen und arbeiten diese nacheinander ab. So können mehrere zu prüfende Konfigurationen definiert und automatisch abgearbeitet werden. Neben dem trainierten Modell mit der geringsten Fehlerrate gibt `run_benchmarks` außerdem eine Statistik zurück, welche als `*.csv`-Datei gespeichert werden kann. Anhand dieser ermittelten Daten können neue Benchmarks erstellt bzw. eine möglichst gute Konfiguration ermittelt werden.

4.3.3 Auswahl der Netzart

Bei dem Betonverteiler besteht die Schwierigkeit darin, die Geschwindigkeit des Verteilers und die Drehzahl der Förderschnecken an die Betonkonsistenz anzupassen. Die bisher eingesetzte Regelung berücksichtigt bei der Berechnung der genannten Regelgrößen (Drehzahl und Geschwindigkeit) den tatsächlichen Betonaustrag pro

Weg in einem zurückliegenden Zeitintervall (*Istwert* - siehe Abschnitt 2.1.3). Damit ist die Regelung adaptiv und kann den Prozess für unterschiedliche Betonkonsistenzen regeln. Wie in Abschnitt 2.2.3 erläutert, eignet sich ein rekurrentes Neuronales Netz, um eine solche zeitliche Abhängigkeit abzubilden.
Für die Wahl einer geeigneten Netzart und -größe wurden zwei Prototypen entwickelt und evaluiert. Sie unterscheiden sich sowohl in der Netzstruktur als auch dem Lernverfahren und sind in Abschnitt 4.3.4 und 4.3.5 beschreiben.

4.3.4 Prototyp 1

Wie die konventionelle Regelung besitzt auch der erste Prototyp die Eingänge für Soll- und Istwert sowie je einen Ausgang für die Schneckendrehzahl r und die Geschwindigkeit v (siehe Abbildung 4.6). Wie bei der Streckensimulation beschrieben, setzt sich der Istwert aus diesen beiden Regelgrößen und einigen Simulationsparametern zusammen. Die Netzstruktur entspricht einem Jordan-Netz, da die Ausgabewerte in je einem Kontextneuronen zwischengespeichert werden und im nächsten Simulationsschritt als zusätzliche Eingangswerte für die versteckten Neuronen dienen. Dadurch ist das Netz in der Lage vorherige Regelgrößen zu berücksichtigen und somit zeitliche Zusammenhänge herzustellen. Die folgende Abbildung zeigt das beschriebene Netz in seiner zeitlich auseinandergefalteten Form.

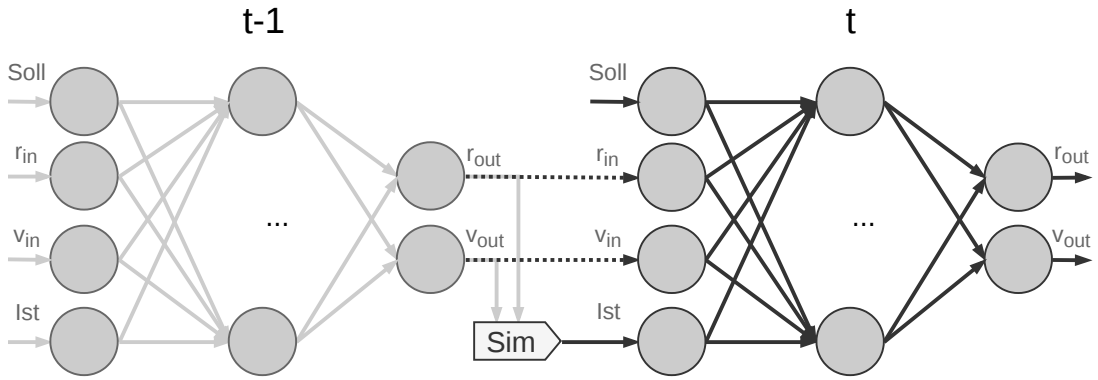

Abbildung 4.6: Netzarchitektur des ersten Prototyps für den aktuellen Zeitpunkt t und den vorherigen $t-1$. r steht für die Schneckendrehzahl und v für die Geschwindigkeit des Betonverteilers.

Die Kontextneuronen r_{in} und v_{in} befinden sich in Abbildung 4.6 auf der gleichen Ebene wie die Eingansneuronen. Die Rückkopplung dieses Netzes entsteht durch die gepunkteten Verbindungen, welche feste Gewichte mit dem Wert 1 haben. Anders ausgedrückt ist der Ausgang des Netzes zu $t-1$ identisch mit den beiden Eingangs-

bzw. Kontextneuronen zu dem aktuellen Zeitpunkt t. Noch wurde die Anzahl der Neuronen in der versteckten Schicht nicht ermittelt, weswegen beispielhaft zunächst zwei versteckte Neuronen eingezeichnet sind.

Wie auch das Jordan-Netz kann das hier beschriebenen Netz mittels *Backpropagation Trough Time* trainiert werden. Um den Unterschied der Lernverfahren zu verdeutlichen, wurde für dieses Modell Backpropagation im Batch-Verfahren aber ohne den zeitlichen Aspekt verwendet.

Das Julia-`Struct` für dieses Netz ist wie folgt aufgebaut:

Listing 4.1: Neuronales Netz von Prototyp 1

```
1  mutable struct NN
2      dims    # layer dimensions
3      W_ij    # input -> hidden layer
4      W_jk    # hidden -> output layer
5      o_js    # hidden layer output
6      o_ks    # output layer output
7
8      function NN(dims::Array)
9          W_ij = randn(dims[1], dims[2]).*(1/sqrt(dims[1]))
10         ...
11         new(dims, W_ij, W_jk, o_js, o_ks)
12     end
13 end
```

Das Netz wird durch eine Liste (`dims`) initialisiert, welche die Größen aller drei Schichten enthält (siehe Zeile 8 in Listing 4.1). Die Initialisierung der Verbindungsgewichte erfolgt anschließend (Zeile 9) nach der im Grundlagenkapitel vorgestellten Gleichung (2.27). Die Variable `o_js` dient als Zwischenspeicher, um die Werte nach einer Sequenz für die Backpropagation zur Verfügung zu haben. Die Werte, welche in `o_ks` gespeichert werden, sind die Eingabe für die Kontextneuronen im nächsten Zeitschritt.

Bestimmung weiterer Netzparameter

In diesem Abschnitt wird die gewählte Netzstruktur überprüft und die für den Anwendungsfall passenden Parameter wie Neuronenanzahl und Lernrate ermittelt.

Bei Anwendung der Faustregeln aus Abschnitt 3.3.2 für die Ermittlung der Anzahl von versteckten Neuronen, ergibt sich bei $N_{in} = 4$ ein Ausgangswert zwischen 2 und 7 Neuronen. Um die geeignete Anzahl und Größe der versteckten Schichten zu ermitteln, wurden mehrere Versuche durchgeführt.

Die erste Versuchsreihe diente zur Ermittlung der Netzgröße. Dazu wurden 100000 Sequenzen der Länge 100 als Trainingsdaten zufällig erzeugt und eine Lernrate $\alpha = 0.02$ gewählt. Diese Lernrate hatte sich bereits während der Entwicklung als gute Wahl erwiesen. Zur Durchführung der Messungen wurden Benchmarks für Netze mit unterschiedlich großen versteckten Neuronenschichten angelegt und an die Funktion `run_benchmarks()` in Datei `benchmark_jordan.jl` übergeben.

Trainingsbeispiele:	100000	Sequenzlänge:	100
Versuche pro Konf.:	20		

Input	Hidden	Output	Lernrate	min. Fehler	max. Fehler	Mittlerer Fehler
2 (4)	3	2	0,02	0,637	0,776	0,696
2 (4)	6	2	0,02	**0,610**	0,723	0,673
2 (4)	9	2	0,02	0,626	0,731	0,679
2 (4)	12	2	0,02	0,629	0,724	0,668
2 (4)	15	2	0,02	0,632	**0,713**	0,660
2 (4)	18	2	0,02	0,617	0,750	**0,657**
2 (4)	21	2	0,02	**0,613**	0,724	0,658
2 (4)	24	2	0,02	0,622	0,721	0,671
2 (4)	27	2	0,02	**0,611**	0,722	0,667
2 (4)	30	2	0,02	0,635	0,715	0,666

Tabelle 4.5: Ermittlung der Anzahl von versteckten Neuronen. Die Zahl in Klammern entspricht der Eingangsneuronen inkl. der Kontextneuronen.

Mit jeder Konfiguration wurden 20 Versuche durchgeführt und der niedrigste, der höchste sowie der durchschnittliche Fehler berechnet.
Um die Vergleichbarkeit der Ergebnisse zu gewährleisten, wurde ein Testdatensatz mit 1000 zufälligen Sequenzen erzeugt und für jeden Versuch der mittlere Fehler des Netzes anhand dieser Daten bestimmt. Der mittlere Fehler in der Tabelle ist der Durchschnitt über alle 20 Versuche mit der entsprechenden Konfiguration.

Anhand der ermittelten Daten ist erkennbar, dass bei einer Lernrate von 0,02 ein Netz mit 6 versteckten Neuronen den kleinsten minimalen Fehler liefert. Allerdings sind die Abweichungen zwischen den Konfigurationen sehr gering. So erzielte ein Netz mit 27 versteckten Neuronen nahezu den gleichen Fehler wie das Netz mit dem besten Ergebnis. Bei dem mittleren Fehler ist allerdings die Tendenz zu erkennen, dass ein Netz mit 18 Neuronen durchschnittlich am weitesten an den Sollwert heranreicht.
Ein Grund für diese geringen Unterschiede ist die dynamische Initialisierung der Kantengewichte. Bei vorhergehenden Versuchsdurchläufen mit einem festen Intervall für die Kantengewichte über alle Netzgrößen hinweg, wurden deutlich größere Unterschiede zwischen dem minimalen und maximalen Fehler festgestellt (siehe `Benchmarks/Jordan.ods` auf dem beiliegenden Datenträger).
Weiterhin besteht die Möglichkeit, dass der Lernprozess zu kurz gewählt wurde, und eigentlich noch niedrigere Fehlerwerte möglich sind. Um die optimale Lernrate zu ermitteln, folgten weitere Versuche unter gleichbleibenden Rahmenbedingungen, mit dem Unterschied, dass die Netzgröße fest und die Lernrate variabel war.

Trainingsbeispiele:	100000	Sequenzlänge:	100
Versuche pro Konf.:	10		

Input	Hidden	Output	Lernrate	min. Fehler	max. Fehler	Mittlerer Fehler
2 (4)	21	2	0,01	0,768	0,821	0,795
2 (4)	21	2	0,0125	0,685	0,753	0,728
2 (4)	21	2	0,015	0,654	0,748	0,699
2 (4)	21	2	0,0175	0,623	0,726	0,673
2 (4)	21	2	0,02	0,613	0,724	0,658
2 (4)	21	2	0,0225	0,618	0,708	0,667
2 (4)	21	2	0,025	0,617	0,693	0,658
2 (4)	21	2	0,0275	**0,594**	**0,679**	**0,655**
2 (4)	21	2	0,03	0,595	0,698	0,658

Tabelle 4.6: Versuche zur Bestimmung der optimalen Lernrate für Prototyp 1

Bei der Ermittlung der Lernrate fällt das Ergebnis deutlicher aus. Je mehr die Lernrate steigt, desto geringer wird der Fehler. Dies ist ein weiteres Indiz dafür, dass der Lernvorgang möglicherweise noch zu kurz war.
Da sich bei einem Jordan-Netz mit steigender Anzahl der versteckten Neuronen die Anzahl der Eingangsneuronen nicht ändert, ist es im Vergleich zu einem Elman-Net

in seiner Komplexität eingeschränkter. Die beiden Kontextneuronen in der Eingangsschicht, welche die zeitlichen Abhängigkeiten ermöglichen, begrenzen gleichzeitig die Anzahl der Verbindungen zu vorhergehenden Netzzuständen.
Möglicherweise ist diese Netzstruktur für die in dieser Arbeit erstellte Simulation ausreichend, würde aber bei der Berücksichtigung weiterer Aspekte an seine Grenzen kommen. Aus diesem Grund wurde ein zweiter, besser skalierbarer Prototyp entwickelt.

4.3.5 Prototyp 2

Dieser zweite Prototyp entspricht im Gegensatz zu dem Ersten, eher dem von ELMAN vorgestellten Netz. Wie in Abbildung 4.7 erkennbar, sind bei diesem Netz die Neuronen der versteckten Schicht zum Zeitpunkt $(t-1)$ mit denen der aktuellen (t) versteckten Schicht verbunden. Eine Erhöhung der Neuronenanzahl dieser Schicht würde auch die Anzahl der zeitlichen Verbindungen (gepunktete Linien) erhöhen, wodurch zwischen zwei Zeitschritten mehr Informationen übertragen werden können.

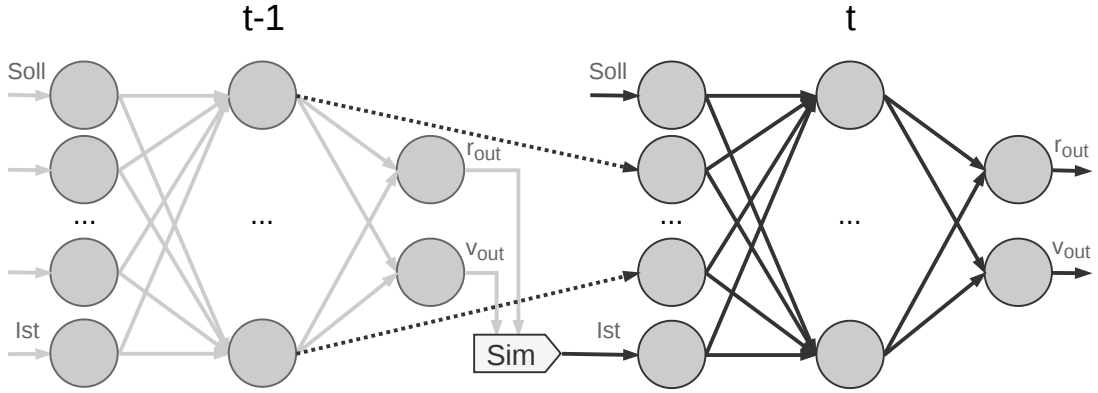

Abbildung 4.7: Netzarchitektur des zweiten Prototyps für den aktuellen Zeitpunkt t und den vorherigen $t-1$. r steht für die Schneckendrehzahl und v für die Geschwindigkeit des Betonverteilers.

Als Trainingsverfahren kommt bei diesem Netz *Backpropagation Through Time* mit dem Batch-Lernverfahren zum Einsatz. Im Vergleich zu dem ersten Prototyp gibt es für die zeitliche Rückkopplung eine separate Gewichtsmatrix (vgl. `W_jj` in Listing 4.2).

Listing 4.2: Neuronales Netz von Prototyp 2

```
mutable struct NN
    dims
    W_ij    # input -> hidden layer
    W_jj    # previous hidden -> current hidden layer
    W_jk    # hidden -> output layer
    o_js    # hidden layer output

    function NN(dims::Array)
        ...
    end
end
```

Die Initialisierung entspricht der des ersten Prototyps und wird für die Verbindungen ebenfalls nach Gleichung 2.27 errechnet.

Bestimmen der Netzparameter

Auch bei diesem Netz wurden die Benchmarks für die Ermittlung von Neuronenanzahl und Lernrate mit 100000 zufällig generierten Trainingsdaten der Länge 100 durchgeführt. Die Ergebnisse in der folgenden Tabelle wurden mit einer Lernrate von $\alpha = 0,01$ erzeugt.

Trainingsbeispiele:	100000	Sequenzlänge:	100
Versuche pro Konf.:	20		

Input	Hidden	Output	Lernrate	min. Fehler	max. Fehler	Mittlerer Fehler
2 (5)	3	2	0,01	0,699	2,002	1,274
2 (8)	6	2	0,01	0,662	1,425	0,776
2 (11)	9	2	0,01	0,596	0,877	0,699
2 (14)	12	2	0,01	0,606	1,003	**0,665**
2 (17)	15	2	0,01	0,577	2,926	0,743
2 (20)	18	2	0,01	0,577	**0,793**	0,683
2 (23)	21	2	0,01	**0,541**	2,927	0,793
2 (26)	24	2	0,01	0,563	2,929	1,234
2 (29)	27	2	0,01	0,554	2,925	0,917
2 (32)	30	2	0,01	0,565	2,927	1,107

Tabelle 4.7: Ermittlung der geeigneten Netzparameter für den zweiten Prototypen. Die Zahl in Klammern entspricht der Eingangsneuronen inkl. der Kontextneuronen.

Auffällig ist bei diesen Ergebnissen, dass es einen größeren Abstand zwischen minimalem und maximalem Fehler gibt als bei Prototyp 1. Die hohen Werte bei den maximalen Fehlern stammen aber in der Regel nur von einem Versuch, weswegen der mittlere Fehler vergleichsweise niedrig bleibt. Diese vereinzelten, hohen Fehlerwerte sorgen dafür, dass der mittlere Fehler bei diesem Prototyp eine geringere Aussagekraft hat.
Bei diesen Messwerten stellt sich heraus, dass ungefähr 21 Neuronen in der versteckten Schicht in Kombination mit einer Lernrate von 0,01 eine gute Kombination ist.

Ob sich das Ergebnis mit einer anderen Lernrate weiter verbessern lässt ermittelte der folgende Benchmark.

Trainingsbeispiele:	100000	Sequenzlänge:	100
Versuche pro Konf.:	10		

Input	Hidden	Output	Lernrate	min. Fehler	max. Fehler	Mittlerer Fehler
2 (23)	21	2	0,005	1,037	1,681	1,318
2 (23)	21	2	0,0075	0,865	**1,096**	1,013
2 (23)	21	2	0,01	0,541	2,927	**0,793**
2 (23)	21	2	0,0125	0,666	2,923	1,205
2 (23)	21	2	0,015	0,526	3,235	1,954
2 (23)	21	2	0,0175	**0,479**	3,192	2,403
2 (23)	21	2	0,02	0,534	2,922	2,484
2 (23)	21	2	0,0225	1,989	3,181	2,855
2 (23)	21	2	0,025	0,539	3,206	2,345

Tabelle 4.8: Versuche zur Bestimmung der optimalen Lernrate für Prototyp 2

In Bezug auf den mittleren Fehler war die ursprünglich gewählte Lernrate von 0,01 eine gute Wahl. Allerdings lieferte die Lernrate 0,0175 das beste Einzelergebnis.

Kapitel 5

Ergebnisse

Anhand der Implementierung der Simulation sowie der Erstellung von zwei Netzprototypen konnte der vorgestellte Ansatz fast vollständig nachvollzogen werden. Da die Inbetriebnahme den Rahmen dieser Arbeit überstiegen hätte, bleibt die Umsetzung des fünften Schrittes unüberprüft.
Im Folgenden werden die Ergebnisse der Arbeit zusammengefasst und diskutiert.

5.1 Simulation

Ein Ergebnis dieser Arbeit ist die in Simulink modellierte Simulation des Betonverteilers und des Regelkreises.
Die Regelung wurde auf Grundlage der Diplomarbeit von K. Heienbrok angefertigt und um Modernisierungen bzw. Details wie den Filter des Istwerts oder die Anhebung des Regelsignals (*shift*) erweitert [15].
In die Simulation des Betonverteilers flossen außerdem 9 von 14 identifizierten Aspekten ein (Abschnitt 4.1.3). Die Vergleichbarkeit von Simulation und Realität ist aufgrund mangelnder Daten der realen Regelung nur beschränkt gegeben. Die mit der Simulation erzeugten Soll- und Istwertkurven (Abbildung 5.1) ähneln aber zumindest den Auszügen der realen Kurven wie der in Abbildung 4.1 gezeigten.

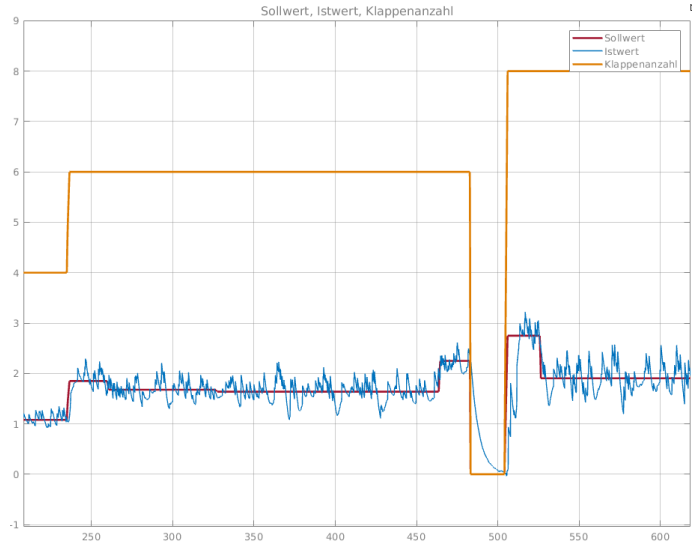

Abbildung 5.1: Ausschnitt eines simulierten Betoniervorgans. Dargestellt sind die Istkurve (blau), die Sollkurve (rot) und die Anzahl der geöffneten Klappen (orange)

Trotz, dass bei der Erstellung der Simulation nicht alles Aspekte modelliert wurden, ähneln beispielsweise die Gewichtsschwankungen den von K. HEIENBROK beschriebenen [15]. Bei einem detaillierteren Vergleich der simulierten Soll- und Istwertkurven mit denen des realen Betoniervorgangs sind allerdings Unterschiede erkennbar. So ist die simulierte Regelung selbst bei hohen P-Anteilen der PI-Regler kaum in Schwingungen zu versetzen. Ein Grund dafür kann die fehlende Komplexität der Streckensimulation sein, wodurch die in der Realität entstehenden physikalischen Schwingungen in dem Modell nicht auftreten.

Diese Differenzen sollten vor einem Einsatz des damit Trainierten Netzes in der Praxis noch behoben werden. Für das Ziel dieser Arbeit, zu untersuchen, ob und wie Neuronale Netze für Regelungsprozesse eingesetzt werden können, ist dieser Unterschied allerdings vernachlässigbar.

5.2 Vergleich der Prototypen

Durch die Erstellung von zwei Prototypen konnten Erfahrungen bei der Modellierung und dem Training Neuronaler Netze gesammelt werden. Wie zu erwarten ist das Jordan-Netz etwas weniger leistungsfähig, als der zweite Prototyp. Dieser konnte bei der gleichen Anzahl von Trainingsbeispielen und einer niedrigeren Lernrate ein besseres Ergebnis erreichen. Die Ursache dafür ist einerseits die strukturbedingte Beschränkung des Jordan-Netzes, dass es nur so viel Rückkopplungen wie Aus-

gangsneuronen gibt und andererseits hat auch das vereinfachte Lernverfahren einen Einfluss auf das Ergebnis.
Unabhängig von dem Lernverfahren konnte aber bei dem zweiten vorgestellten Netz eine dynamischere Struktur erstellt werden, welche auch noch mit komplexeren Simulationen verwendbar und damit besser für den Produktiveinsatz geeignet ist.

Die erstellten Netze kamen in einem simulierten Regelkreislauf zum Einsatz. Auf die daraus entstandenen Resultate wird im folgenden Abschnitt eingegangen.

5.3 Neuronale Regelung

Um die Ergebnisse der erstellten neuronalen Regelung zu beurteilen, ist es naheliegend, diese in Relation zu der simulierten konventionellen Regelung zu setzen.

Das Vergleichskriterium bildete die in dieser Arbeit bereits gezeigte reale Sollkurve (siehe Tabelle A.1). Ein trainiertes Netz des zweiten Prototyps erzeugte bei dem entsprechenden Betoniervorgang über die Distanz von rund 9m eine Abweichung zu dem Sollwert von ca. 4,58. Für den gleichen Vorgang kam die simulierte Regelung auf eine Abweichung von 174,1.
Der Fehler wurde nach folgender Gleichung ermittelt:

$$Abweichung = \sum_{t}(|Soll - Ist|) \tag{5.1}$$

Über den kompletten Betoniervorgang wurde also für jeden Simulationsschritt die absolute Differenz zwischen Soll- und Istwert aufsummiert.

Die neuronale Regelung wurde zuvor mit 100000 zufälligen Sollkurven der Länge 100 trainiert. Die in diesem Versuch verwendete Kurve hatte das Netz vorher nicht gelernt. Außerdem ist diese Testsequenz mit 1000 Datenpunkten deutlich größer als die Trainingsbeispiele.
Dieser Versuch zeigt die Generalisierungsfähigkeit Neuronaler Netze und legt nahe, dass diese tatsächlich auch für Regelungsprozesse eingesetzt werden können. Möglicherweise lässt sich die Abweichung der simulierten konventionellen Regelung duch das Anpassen einiger Stellgrößen weiter reduzieren. Außerdem ist es fraglich, ob sich dieses Ergebnis ohne Weiteres in die Realität übertragen lässt. Die folgende Grafik zeigt, die Regelkurve des Neuronalen Netzes.

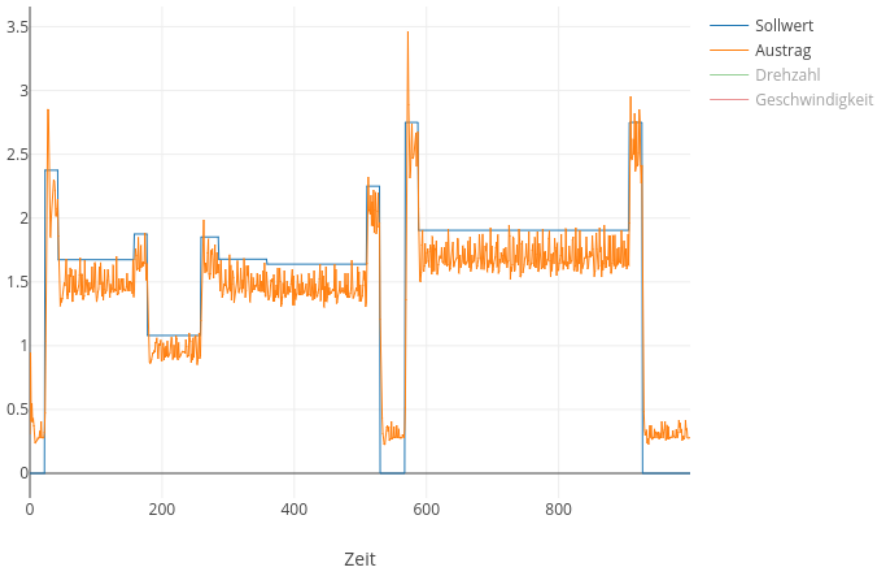

Abbildung 5.2: Soll- und Istwertkurve der neuronalen Regelung

Es ist erkennbar, dass der Austrag zu einem großen Teil unter dem Sollwert zurückbleibt, was durch weitere Anpassungen der Trainingsdaten oder Netzparameter behoben werden kann.

Kapitel 6

Zusammenfassung und Ausblick

Ausgehend von den Grundlagen Neuronaler Netze und bereits vorhandener Möglichkeiten diese in Regelungssysteme zu integrieren, wurde in dieser Arbeit ein Konzept erarbeitet, welche Schritte für die Erstellung einer neuronalen Regelung notwendig sind.
Da Neuronale Netze zunächst für eine Aufgabenstellung trainiert werden müssen, beginnt die Entwicklung der Regelung mit der Beschaffung von Trainingsdaten. Diese können beispielsweise aus einem vorhandenen System extrahiert oder mit einer Simulation erzeugt werden.
Es folgt die Auswahl einer geeigneten Regelarchitektur, die wiederum stark von den vorhandenen Trainingsdaten abhängt.
Wenn die Art der Regelung bestimmt wurde, ist es wichtig, das für den Anwendungsfall am besten geeignete Neuronale Netz zu wählen, um möglichst gute Ergebnisse zu erhalten.
Einen zentralen Einfluss auf den Erfolg hat das Training des Netzes, wo es ebenfalls je nach Netzart verschiedene Verfahren gibt.
Ist ein Netz schließlich gut genug trainiert, kann es weiter getestet und die neuronale Regelung in Betrieb genommen werden.

Anhand des Anwendungsbeispiels des Betonverteilers wurde gezeigt, wie sich dieses Konzept anwenden lässt. Da die Gewinnung von Daten aus einem Produktivsystem nur sehr eingeschränkt möglich war, wurde eine Simulation des Betonverteilers und der dazugehörigen Regelung erstellt. Anhand dieser Simulation konnten schließlich zwei Prototypen trainiert und evaluiert werden. Während der erste Prototyp eine vergleichsweise einfache Struktur hat und auch in seiner Fähigkeit zu skalieren eingeschränkt ist, ist das zweite Netz leistungsfähiger und auch dazu in der Lage

komplexere Sachverhalte abzubilden.

Die entwickelte neuronale Regelung konnte allerdings noch nicht an einer realen Maschine getestet werden, da dies den Rahmen dieser Arbeit überstiegen hätte. Es zeichnete sich im Vergleich zwischen konventioneller und neuronaler Regelung ab, dass Neuronale Netze dazu in der Lage sind, für Regelungsaufgaben eingesetzt zu werden.

Aufbauen auf den hier gelegten Grundlagen wäre es denkbar, die Entwicklung weiterzuführen und die Regelung in der Praxis zu testen. Dazu ist zunächst eine weitere Verfeinerung der Simulation empfehlenswert. Je realistischer die Trainingsdaten sind, desto besser kann das Netz letztendlich in einem realen System funktionieren. Außerdem wurde in dieser Arbeit auf die Verwendung von Machine-Learning Frameworks verzichtet, um Neuronale Netze möglichst umfassend zu verstehen und anpassen zu können. Für weiteres Training sollte in Betracht gezogen werden, dass Frameworks meist deutlich effizienter arbeiten.

Abschließend lässt sich sagen, dass das Thema künstliche Intelligenz auch in der Regelungstechnik eine Berechtigung hat und beispielsweise Neuronale Netze für solche Zwecke geeignet sind. Vor allem, wenn es für ein Problem keine optimale Lösung gibt und konventionelle Methoden ausgeschöpft sind, sollte der Einsatz von Techniken der künstlichen Intelligenz in Betracht gezogen werden.

Anhang A

Weitere Informationen

A.1 Steigende Anzahl von Artikeln zum Thema Künstliche Intelligenz

Seit dem Jahr 2015 nimmt die Anzahl von veröffentlichten Artikeln in der Kategorie „Künstliche Intelligenz“ deutlich zu. Betrachtet wurden die drei (online)-Zeitschriften heise online, golem.de und SPIEGEL ONLINE.

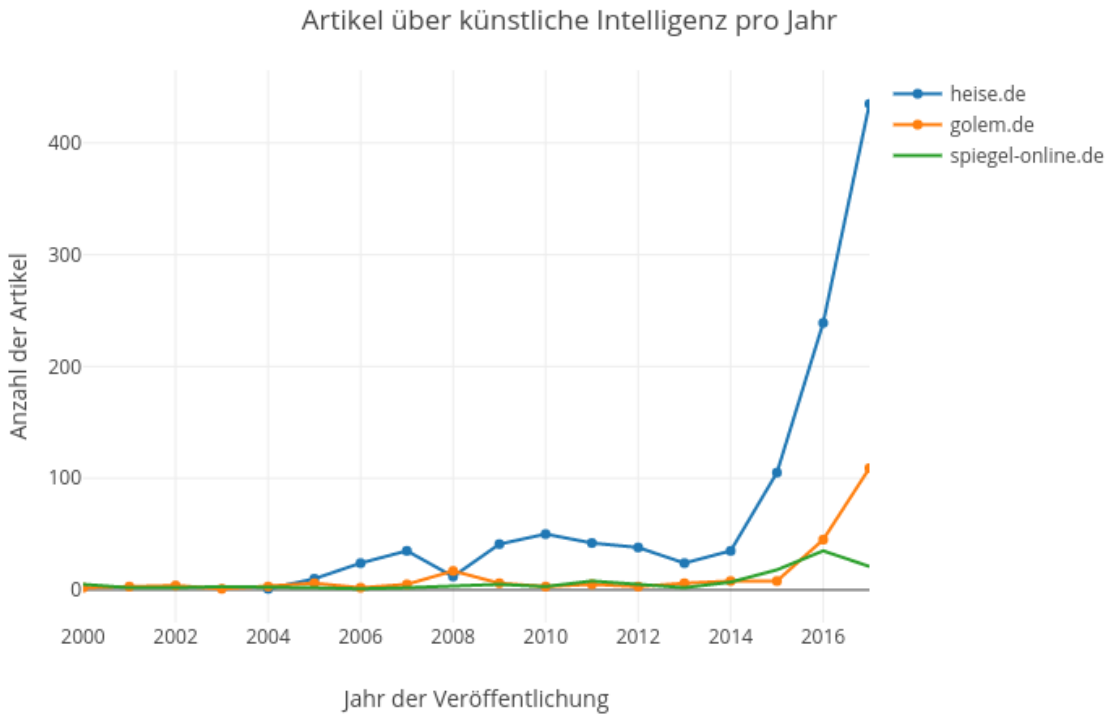

Abbildung A.1: Anzahl der Artikel aus der Kategorie „Künstliche Intelligenz“ in den Jahren 2000-2017

Weitere Informationen zu dieser Statistik: `https://github.com/JoramM/news-scraper.jl`

A.2 Realer Betoniervorgang

X-Pos in cm	Strecke in cm	Gewicht in kg	Klappen	Sollwert
23	0	0	0	0,000
43	20	47,5	6	2,375
158	115	192,5	6	1,674
178	20	37,5	6	1,875
259	81	87,5	4	1,080
286	27	50	6	1,852
359	73	122,5	6	1,678
510	151	247,5	6	1,639
530	20	45	6	2,250
568	38	0	0	0,000
588	20	55	8	2,750
907	319	607,5	8	1,904
927	20	55	8	2,750

Tabelle A.1: Auszug eines realen Betoniervorgangs

Abbildungsverzeichnis

Algorithmenverzeichnis

Literaturverzeichnis

[1] Anderson, John Robert: *Kognitive Psychologie*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[2] Architektur-Lexikon: *Bewehrung*. https://www.architektur-lexikon.de/cms/lexikon/35-lexikon-b/264-bewehrung.html, 2018. [Online; Abgerufen am 20.02.2018].

[3] Aruba, S. Boraan und Jesús Fernández-Villaverde: *A Comparison of Programming Laguages in Economics*. NBER Working Paper Series, 20643(June):1–20, 2014.

[4] Azevedo, Frederico A.C., Ludmila R.B. Carvalho, Lea T. Grinberg, José Marcelo Farfel, Renata E.L. Ferretti, Renata E.P. Leite, Wilson Jacob Filho, Roberto Lent und Suzana Herculano-Houzel: *Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain*. The Journal of Comparative Neurology, 513(5):532–541, apr 2009.

[5] Barto, Andrew G.: *Connectionist Learning for Control*. In: *Neural Networks for Control*, Kapitel 1, Seiten 5–58. Massachusetts Institute of Technology, 1990.

[6] Bezanson, Jeff, Stefan Karpinski, Viral B. Shah und Alan Edelman: *Julia: A Fast Dynamic Language for Technical Computing*. Seiten 1–27, 2012.

[7] Cybenko, G: *Approximation by superpositions of a sigmoidal function*. Mathematics of Control, Signals and Systems, 2(4):303–314, 1989.

[8] Elman, Jeffrey L.: *Finding Structure in Time*. Cognitive Science, 14(2):179–211, 1990.

[9] Ertel, Wolfgang: *Grundkurs Künstliche Intelligenz*. 2016.

[10] Goasduff, Laurence: *2018 Will Mark the Beginning of AI Democratization*. https://www.gartner.com/smarterwithgartner/2018-will-mark-the-beginning-of-ai-democratization/, 2017. [Online; Abgerufen am 02.12.2017].

[11] Hafner, Sigrid (Hrsg.): *Neuronale Netze in der Automatisierungstechnik*. R. Oldenbourg Verlag GmbH, München, 1994.

[12] Hare, Jim: *The biggest roadblock to AI adoption is a lack of skilled workers*. https://venturebeat.com/2017/11/04/the-biggest-roadblock-in-ai-adoption-is-a-lack-of-skilled-workers/, 2017. [Online; Abgerufen am 04.04.2018].

[13] Heaton, Jeff: *The Number of Hidden Layers*. http://www.heatonresearch.com/2017/06/01/hidden-layers.html, 2017. [Online; Abgerufen am 19.03.2018].

[14] Hebb, Donald Olding: *The organization of behavior: A neuropsychological theory*. Wiley, New York, 1949.

[15] Heienbrok, Knut: *Entwicklung und Inbetriebnahme einer Mehrgrößendosierregelung für ein Betonfertigteilwerk*, 1997.

[16] Hochreiter, Josef: *Untersuchungen zu dynamischen neuronalen Netzen*, 1991.

[17] Hochreiter, Sepp und Jürgen Schmidhuber: *Long Short-Term Memory*. Neural Computation, 9(8):1735–1780, 1997.

[18] Hopfield, J J: *Neural networks and physical systems with emergent collective computational abilities*. Proceedings of the National Academy of Sciences, 79(8):2554–2558, 1982.

[19] Ingenieurbüro Dr. Kahlert: *Datenaustausch zwischen BORIS und Simulink über UDP*. http://www.kahlert.com/datenaustausch-zwischen-boris-und-simulink-ueber-udp/, 2018. [Online; Abgerufen am 23.03.2018].

[20] Ingenieurbüro Dr. Kahlert: *Preislisten*. http://www.kahlert.com/preislisten/, 2018. [Online; Abgerufen am 23.03.2018].

[21] Ingenieurbüro Dr. Kahlert: *WinFACT Übersicht*. http://www.kahlert.com/winfact-uebersicht/, 2018. [Online; Abgerufen am 23.03.2018].

[22] Isermann, Mario: *Kognitive Regelung komplexer Produktionsprozesse*. Doktorarbeit, RWTH Aachen, 2014.

[23] Jordan, Michael I.: *Attractor dynamics and parallelism in a connectionist sequential machine*. In: *Proc. of the Eighth Annual Conference of the Cognitive Science Society*, Seiten 531–546, 1986.

[24] Kapur, Rohan: *Rohan #4: The vanishing gradient problem*. 2016. [Online; Abgerufen am 06.04.2018].

[25] Karpathy, Andrej: *Neural Networks Part 1: Setting up the Architecture*. http://cs231n.github.io/neural-networks-1, 2017. [Online; Abgerufen am 07.04.2018].

[26] Karpathy, Andrej: *Neural Networks Part 2: Setting up the Data and the Loss*. http://cs231n.github.io/neural-networks-2, 2017. [Online; Abgerufen am 07.04.2018].

[27] Kriesel, David: *Ein kleiner Überblick über Neuronale Netze*. 2007. http://www.dkriesel.com/science/neural_networks.

[28] Krizhevsky, Alex, Ilya Sutskever und Geoffrey E Hinton: *ImageNet Classification with Deep Convolutional Neural Networks*. Advances In Neural Information Processing Systems, Seiten 1–9, 2012.

[29] Kröger, Bernd J.: *Neuronale Modellierung der Sprachverarbeitung und des Sprachlernens*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2018.

[30] Kruse, Rudolf, Christian Borgelt, Christian Braune, Frank Klawonn, Christian Moewes und Matthias Steinbrecher: *Computational Intelligence: Eine methodische Einführung in Künstliche Neuronale Netze, Evolutionäre Algorithmen, Fuzzy-Systeme und Bayes-Netze*. 2015.

[31] Lämmel, Uwe und Jürgen Cleve: *Künstliche Intelligenz*. Carl Hanser Verlag GmbH & Co. KG, München, oct 2008.

[32] LeCun, Y., B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard und L. D. Jackel: *Backpropagation Applied to Handwritten Zip Code Recognition*. Neural Computation, 1(4):541–551, Dec 1989.

[33] LeCun, Yann, Leon Bottou, Genevieve B. Orr und Klaus Robert Müller: *Efficient BackProp*. Seiten 9–50. 1998.

[34] McCulloch, Warren S und Walter Pitts: *A logical calculus of the ideas immanent in nervous activity.* The bulletin of mathematical biophysics, 5(4):115–133, 1943.

[35] Neumerkel, Dietmar und Friedrich Lohnert: *Modellierung und Regelung mit Neuronalen Netzen.* In: *Neuronale Netze in der Automatisierungstechnik*, Seiten 31–47. R. Oldenbourg Verlag München Wien, Berlin, 1994.

[36] Olah, Christopher: *Understanding LSTM Networks.* http://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015. [Online; Abgerufen am 07.03.2018].

[37] Puget, Jean Francois: *The Most Popular Language For Machine Learning Is ...* https://www.ibm.com/developerworks/community/blogs/jfp/entry/What{_}Language{_}Is{_}Best{_}For{_}Machine{_}Learning{_}And{_}Data{_}Science?lang=en, 2016. [Online; Abgerufen am 02.04.2018].

[38] Rashid, Tariq: *Neuronale Netze selbst programmieren.* Heidelberg : O'Reilly, 2017.

[39] Rich, Elaine: *Artificial Intelligence.* McGraw-Hill Inc., New York, 1983.

[40] Rumelhart, David E., Geoffrey E. Hinton und Ronald J. Williams: *Learning representations by back-propagating errors.* Nature, 323(6088):533–536, 1986.

[41] Scilab Enterprises S.A.S: *About Scilab.* http://www.scilab.org/en/scilab/about, 2018. [Online; Abgerufen am 23.03.2018].

[42] Scilab Enterprises S.A.S: *API Scilab.* https://help.scilab.org/docs/6.0.1/en_US/api_scilab.html, 2018. [Online; Abgerufen am 23.03.2018].

[43] Sopra Steria Consulting: *Potenzialanalyse künstliche intelligenz 2017.* Technischer Bericht, 2017.

[44] SPLENDID RESEARCH GmbH: *Wie verbreitet sind digitale Sprachassistenten in Deutschland?* https://www.splendid-research.com/studie-digitale-sprachassistenten.html, 2017. [Online; Abgerufen am 15.03.2018].

[45] Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever und Ruslan Salakhutdinov: *Dropout: A Simple Way to Prevent*

Neural Networks from Overfitting. Journal of Machine Learning Research, 15:1929–1958, 2014.

[46] Stack Exchange Inc.: *Newest 'matlab' Questions - Stack Overflow.* https://stackoverflow.com/questions/tagged/matlab. [Online; Abgerufen am 23.03.2018].

[47] Stack Exchange Inc.: *Newest 'simulink' Questions - Stack Overflow.* https://stackoverflow.com/questions/tagged/simulink. [Online; Abgerufen am 23.03.2018].

[48] Stanley, Kenneth O. und Risto Miikkulainen: *Efficient Reinforcement Learning Through Evolving Neural Network Topologies.* In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002), 10(4):569–577, 2002.

[49] The MathWorks, Inc.: *Features - MATLAB Compiler SDK.* https://www.mathworks.com/products/matlab-compiler-sdk/features.html. [Online; Abgerufen am 23.03.2018].

[50] The MathWorks, Inc.: *Siemens SIMATIC STEP 7 Support from Simulink PLC Coder.* https://www.mathworks.com/hardware-support/siemens-step-7.html. [Online; Abgerufen am 23.03.2018].

[51] The MathWorks, Inc.: *WinFACT Übersicht.* https://de.mathworks.com/pricing-licensing.html?prodcode=SL. [Online; Abgerufen am 23.03.2018].

[52] The MathWorks, Inc.: *Write and Read Data over TCP/IP Interface.* https://de.mathworks.com/help/matlab/import_export/write-and-read-data-over-the-tcpip-interface.html. [Online; Abgerufen am 23.03.2018].

[53] Wartala, Ramon: *Praxiseinstieg Deep Learning.* O'Reilly, Heidelberg, 2018.

[54] Werbos, Paul: *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* Doktorarbeit, Harvard University, 1974.

[55] Werbos, Paul J.: *Backpropagation Through Time: What It Does and How to Do It.* Proceedings of the IEEE, 78(10):1550–1560, 1990.

[56] Wikimedia Commons: *File:Neuron (deutsch)-1.svg — Wikimedia Commons, the free media repository.* https://commons.wikimedia.org/w/index.php?ti

tle=File:Neuron_(deutsch)-1.svg&oldid=242338943, 2017. [Online; Abgerufen am 20.02.2018].

[57] Zhang, Chiyuan: *Mocha.jl: Deep Learning for Julia.* https://devblogs.nvidia.com/mocha-jl-deep-learning-julia/, 2015. [Online; Abgerufen am 15.03.2018].

[58] Ziegler, John G und Nathaniel B Nichols: *Optimum settings for automatic controllers.* trans. ASME, 64(11), 1942.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Gummersbach, den 12.04.2018

Joram Markert