

Jonas Helmut Wilinski

Eine Anwendung des Reinforcement Learning zur Regelung dynamischer Systeme

Bachelor-Arbeit

Stand: September 2018

© Lehrstuhl für Regelungstechnik
Christian-Albrechts-Universität zu Kiel

Jonas Helmut Wilinski

Eine Anwendung des Reinforcement Learning zur Regelung dynamischer Systeme

Bachelor-Arbeit

Prüfer: Prof. Dr.-Ing habil. Thomas Meurer
Abgabedatum: 08. September 2018

Diese Arbeit ist eine Prüfungsarbeit. Anderweitige Verwendung und die Weitergabe an Dritte, ist nur mit Genehmigung des betreuenden Lehrstuhls gestattet.

Inhaltsverzeichnis

Abstract und Kurzfassung	1
1 Grundlagen der neuronalen Netze	3
1.1 Grundlegende Berechenbarkeitsmodelle	3
1.2 Die biologische Nervenzelle	4
1.3 Das biologische neuronale Netz	6
1.4 Das symmetrische neuronale Netz	7
2 Leaky Integrate and Fire und simulative Modelle neuronaler Netze	9
2.1 Das Leaky Integrate and Fire - Modell	9
2.2 Anwendung auf Modelle neuronaler Netze	11
2.3 Zuverlässigkeit und Limitationen	12
2.4 Implementierung	12
3 Reinforcement Learning - Lernen mit Belohnung	17
3.1 Reinforcement Learning - eine Abwandlung des Deep Learning	17
3.2 Anwendung auf Modelle neuronaler Netze	19
3.3 Verschiedene Suchalgorithmen	19
3.3.1 Q-Learning	20
3.3.2 Gradient Policies	20
3.3.3 Genetische Algorithmen	20
3.3.4 Random Search	20
4 Implementierung des TW-Netzes	21
4.1 Aufbau des Programms	21

4.2	Implementierung der Suchalgorithmen	22
4.2.1	Suchalgorithmus RandomSearch	22
4.2.2	Optimierungsalgorithmus Weights	23
4.2.3	Simulation in der Google Cloud Platform®	24
4.3	Simulationsumgebung: OpenAI Gym	25
4.4	Visualisierung und Auswertung	28
4.5	Sonstige Implementierung	28
4.5.1	Zentraler Ort für Parameter	29
4.5.2	Speicherung von Daten	29
4.5.3	Dateiinspektion	29
5	Performance & Auswertung	31
5.1	Performance implementierter Algorithmen	31
5.2	Limitationen und Alternativen von Algorithmen	32
5.2.1	Analyse bereits bestehender Algorithmen	33
5.2.2	Alternative Such- und Optimierungsalgorithmen	34
5.3	Vergleich zu bestehenden Systemen	34
5.4	Zusammenfassung	34
5.5	Ausblick	34
A	Programmcode Python	35
A.1	LIF-Modell	35
B	Datenblatt Programm: TW-Circuit	37
C	Parameter mit guten Simulationsergebnissen	39
	Literaturverzeichnis	41

Abstract und Kurzfassung

Abstract

English version ... approx. $\frac{1}{2}$ page

Kurzfassung

Diese Bachelorarbeit umfasst den Ansatz, durch *Reinforcement Learning* eine zuverlässige und vergleichbare Regelung von dynamischen Systemen zu erzielen, welche bisher nur durch klassische Regelansätze möglich gewesen ist. Dabei wird im ersten Schritt in Anlehnung an das Tierreich das neuronale Netz des Wurms *C. Elegans* [3] genauer betrachtet, um Rückschlüsse auf die Lernalgorithmen zu erhalten. Die Verschaltung der verschiedenen Neuronen mittels Synapsen lassen sich so exakt mathematisch durch das s.g. *Leaky Integrate and Fire Model* beschreiben und simulativ nachbauen.

Im zweiten Teil wird mittels der Programmiersprache **Python** ein neuronales Netz zur Regelung der Problemstellung des inversen Pendels implementiert und durch Reinforcement Learning Algorithmen trainiert. Hier kommen verschiedene Werkzeuge wie Python Libraries, TensorFlow und andere Hilfsmittel zum Einsatz.

Ziel dieser Bachelorarbeit ist es, eine verlässliche Regelung dynamischer Systeme als Simulation zu erschaffen und diese mit konventionellen Ansätzen der Regelungstechnik qualitativ zu vergleichen.

Kapitel 1

Grundlagen der neuronalen Netze

Dieses Kapitel dient als Einführung in meine Arbeit und zeigt auf, wie im Laufe der Zeit das Konstrukt der neuronalen Netze erforscht und zu Nutze gemacht wurde. Darüber hinaus wird die Notwendigkeit einer Alternative zum klassischen Modell der Rechenmaschine aufgezeigt und genauer beleuchtet. Um die Performance dieser klassischen Automaten bzw. Rechenmaschinen zu testen, werden einige Berechenbarkeitsmodelle vorgestellt. Die einzigartige Umsetzung in Netzwerken neuronaler Nervenzellen ermöglicht es uns, hochkomplexe Aufgaben selbst bei niedrigen Taktfrequenzen durch hohe Parallelität zu bewältigen. Im weiteren Verlauf wird auch auf die Notation und Beschaffenheit neuronaler Netze eingegangen.

Dieser Abschnitt der Bachelorarbeit lehnt sich besonders an die ersten Kapitel der folgenden Bücher an: *R.Rojas - Theorie der neuronalen Netze* [4] und *Gerstner et al. - Neuronal Dynamics* [8]. Weitere Fachartikel werden im Laufe des Kapitels genannt.

1.1 Grundlegende Berechenbarkeitsmodelle

Im Bereich der Berechenbarkeitstheorie (oder auch Rekursionstheorie) werden Probleme auf die Realisierbarkeit durch ein mathematisches Modell einer Maschine bzw. einem Algorithmus untersucht und kategorisiert. Diese Theorie entwickelte sich aus der mathematischen Logik und der theoretischen Informatik. Neuronale Netze bieten hier eine alternative Formulierung der Berechenbarkeit neben den bereits etablierten Modellen an. Es existieren die folgenden fünf Berechenbarkeitsmodelle, welche durch einen mathematischen oder physikalischen Ansatz versuchen, ein gegebenes Problem zu lösen:

- Das mathematische Modell

Die Frage nach der Berechenbarkeit wird in der Mathematik durch die zur Verfügung stehenden Mittel dargestellt. So sind primitive Funktionen und Kompositionsregeln offensichtlich zu berechnen, komplexe Funktionen, welche sich nicht durch primitive Probleme darstellen lassen, jedoch nicht. Durch die *Church-Turing-These*¹ wurden die berechenbaren Funktionen wie folgt abgegrenzt: “Die berechenbaren Funktionen sind die allgemein rekursiven Funktionen.“

- Das logisch-operationelle Modell (Turing Maschine)

¹ Alonzo Church & Alan Turing, 1936

Durch die Turing Maschine ² konnte neben der mathematischen Herangehensweise an Berechenbarkeitsprobleme eine mechanische Methode eingesetzt werden. Die Turing Maschine nutzte ein langes Speicherband, welches nach gewissen Regeln schrittweise Manipuliert wurde. So konnte sie sich in einer bestimmten Anzahl von Zuständen befinden und nach entsprechenden Regeln verfahren.

- Das Computer-Modell

Kurz nach dem bahnbrechenden Erfolg von Turing und Church wurden viele Konzepte für elektrische Rechenmaschinen entworfen. Konrad Zuse entwickelte in Berlin ab 1938 Rechenautomaten, welche jedoch nicht in der Lage waren, alle allgemein rekursiven Funktionen zu lösen. Der Mark I, welcher um 1948 an der Manchester Universität gebaut wurde war der erste Computer, welcher alle rekursiven Funktionen lösen konnte. Er verfügte über die damals etablierte Von-Neumann-Architektur³ und wurde von Frederic Calland Williams erbaut.

- Das Modell der Zellautomaten

John von Neumann arbeitete darüber hinaus auch an dem Modell der Zellautomaten, welches eine hoch-parallele Umgebung bot. Die Synchronisation und Kommunikation zwischen den Zellen stellt sich jedoch als herausfordernde Problemstellung heraus, welche nur durch bestimmte Algorithmen gelöst werden kann. Eine solche Umgebung liefert, wenn richtig umgesetzt, eine enorme Rechenleistung dank Multiprozessorarchitektur selbst bei geringen Taktfrequenzen.

- Das biologische Modell (neuronale Netze)

Neuronale Netze heben sich nun von den vorher beschriebenen Methoden ab. Sie sind nicht sequentiell aufgebaut und können, anders als Zellautomaten, eine hierarchische Schichtenstruktur besitzen. Die Übertragung von Informationen ist daher nicht nur zum Zellnachbarn, sondern im ganzen Netzwerk möglich. Jedoch wird im neuronalen Netz nicht (wie in der Rechenmaschine üblich) ein Programm gespeichert, sondern es muss durch die s.g. Netzparameter erlernt werden. Dieser Ansatz wurde früher durch mangelnde Rechenleistung der konventionellen Computer nicht weiter verfolgt. Jedoch erfahren wir heute immer mehr den Aufwind neuester Lernalgorithmen und Frameworks, die das Arbeiten im Bereich Deep Learning, Artificial Intelligence und adaptives Handeln unheimlich unterstützen und beschleunigen. Weitergehend ist man heute in der Lage, auf dem Gebiet der Biologie Nervensysteme zu analysieren und von Millionen Jahren der Evolution zu profitieren. So können verschiedene neuronale Netze genauestens beschrieben und simuliert werden.

1.2 Die biologische Nervenzelle

Zellen, wie sie in jeder bekannten Lebensform auftreten, sind weitestgehend erforscht und gut verstanden. Wie alle Zellen im Körper bestehen Sie (stark vereinfacht) aus einer Zellmembran, einem Zellskelett und einem Zellkern, welcher die chromosomale DNA und somit die Mehrzahl der Gene enthält. Sie treten im menschlichen Körper in verschiedenen Größen und mit unter-

² Alan Turing, 1936

³ John von Neumann, 1945

schiedlichen Fähigkeiten auf. Neuronale Nervenzellen wurden über die Evolution dahingehend ausgeprägt, dass sie Informationen Empfangen, verarbeiten und entsenden können. Wie in Abb. 1.1 zu sehen, besteht eine Nervenzelle aus drei Bestandteilen: *Dendrit*, *Soma* und *Axon*.

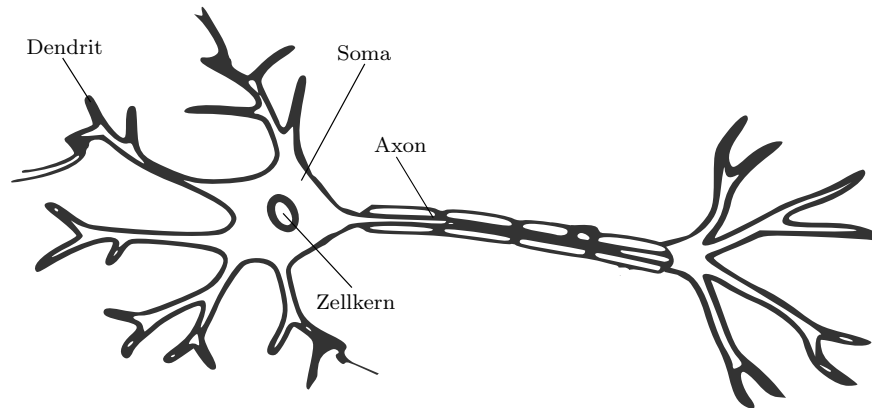


Abb. 1.1: Schematische Darstellung einer Nervenzelle bestehend aus Dendrit, Soma und Axon.

- Dendrit:

Der Dendrit (altgr. 'Baum') dient der Reizaufnahme in der Nervenzelle. Gelangen durch andere Nervenzellen Spannungsspitzen durch vorhandene Synapsen an den Dendrit, leitet dieser die Signale an die Soma weiter.

- Soma:

Die Zellsoma bezeichnet den allg. Körper der Zelle. Es umfasst den plasmatischen Bereich um den Zellkern, ohne die Zellfortsätze wie Dendriten und Axon. Hier findet der Hauptteil des Stoffwechsels statt, alle ankommenden Signale aus den Dendriten werden integrierend verarbeitet und eine Änderung des Membranpotentials findet statt. Empfangene Signale können erregend oder hemmend auf den Summationsprozess wirken (Siehe Kap. x - LIF Modell). Überschreitet das Membranpotential einen gewissen Threshold, so reagiert die Soma und erzeugt einen Spannungsstoß, welcher an das Axon gegeben wird.

- Axon:

Das Axon (altgr. 'Achse') ist ein Nervenzellfortsatz, welcher für die Weiterleitung der Signale von der Soma an die Synapsen und damit an andere Nervenzellen verantwortlich ist.

Verbunden sind Nervenzellen durch s.g. Synapsen, welche den Informationsfluss gewährleisten. Der Informationsfluss geschieht in Synapsen größtenteils chemisch. Bei einem ankommenden Aktionspotential werden Neurotransmitter aus der Zelle ausgeschüttet, welche für einen Ionen-transport verantwortlich sind. Nach Übertragung der chemischen Stoffe über den Synapsenspalt werden diese wieder in ein elektrisches Potential umgewandelt. Diese Synapsen treten zwischen benachbarten Nervenzellen bzw. auf kurzer Distanz auf. Elektrische Synapsen hingegen sind noch weitestgehend unerforscht. Sie dienen als Kontaktstellen und ermöglichen eine Übertragung von Ionen und kleineren Molekülen von einer Zelle zur anderen. Die Signalübertragung entfernter Nervenzellen wird somit synchronisiert. Man bezeichnet sie auch als "Gap-Junctions". Im wei-

teren Verlauf dieser Arbeit werden Synapsen nach Abb. 1.2 dargestellt.

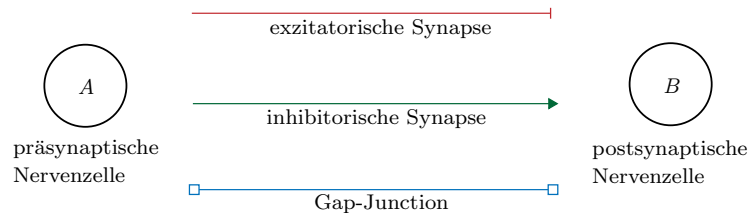


Abb. 1.2: Darstellung von verschiedenen Synapsen-Typen.

Bei chemischen Synapsen ist zwischen exzitatorischen und inhibitorischen Synapsen zu unterscheiden. Erstgenannte agieren als erregende Synapsen und übertragen das Aktionspotential mit positivem Vorzeichen an die postsynaptische Nervenzelle. Inhibitorische Synapsen sind hingegen hemmender Natur und führen das Potential mit einem negativen Vorzeichen, sodass es entsprechend negativ gewichtet in den Integrationsprozess der postsynaptischen Nervenzelle eingeht.

1.3 Das biologische neuronale Netz

Funktionsweisen neuronaler Netze sind bereits gut erforscht und modelliert worden. Besonders das Nervensystem des Wurms *C. Elegans* [1] ist das bisher am besten verstandene Konstrukt in diesem Bereich der neuronalen Forschung. In dieser Arbeit wird insbesondere auf den s.g. *Touch-Withdrawal-Circuit* eingegangen und versucht, eine Implementierung zu schaffen, welche ein dynamisches System erfolgreich regeln kann.

Ausgangspunkt ist das bereits von Lechner et al. [3] graphisch dargestellte neuronale Netz des *C. Elegans*, welches den Berührungs-Reflex des Wurms modelliert. Wird der Wurm einem äußeren

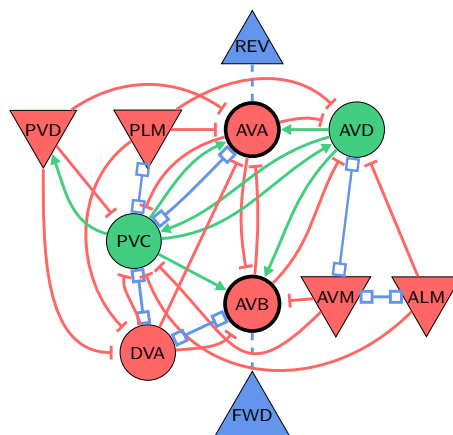


Abb. 1.3: TW-Neuronal-Circuit nach Lechner et al. [3]

Stimulus - sprich einer Berührung - ausgesetzt, so schnell er zurück. Anhand des Schaubilds lässt sich nachvollziehen, was in dem Fall einer Berührung in dem neuronalen Netz geschieht: Die Sensor-Neuronen PVD, PLM, AVM und ALM stellen Rezeptorzellen dar und reagieren auf Berührung. Sie transduzieren in diesem Fall die Berührung in eine neuronal vergleichbare Form als Aktionspotential und übermitteln diese Information durch die gegebenen Synapsen inhibitorisch oder exzitatorisch an die verbundenen internen Nervenzellen. Dieses Potential beträgt je nach gegebener Intensität der Berührung zwischen $-70mV$ (Ruhespannung - keine Berührung) und $-20mV$ (Spike-Potential - maximale Berührungsintensität) und bildet den Aktionsraum $A \in [-70mV, -20mV]$. Die genannten Sensor-Neuronen lassen sich so beliebig einsetzen und stellen bspw. im Experiment des inversen Pendels positive und negative Observationsgrößen dar. Eine beispielhafte Belegung wäre die folgende:

<i>Umgebungsvariable</i>	<i>Typ</i>	<i>Positive Sensor-Neurone</i>	<i>Negative Sensor-Neurone</i>
φ	Observation	PLM	AVM
$\dot{\varphi}$	Observation	ALM	PVD
a	Action	FWD	REV

Im weiteren Verlauf der Bachelorarbeit werden zudem Vor- und Nachteile aufgezeigt, die genannten Sensor-Neuronen mit anderen Observationsgrößen zu belegen.

Interneuronen, wie PVC, AVD, DVA, AVA und AVB sind direkt mit Sensor-Neuronen sowie untereinander durch Synapsen und Gap-Junctions verbunden. In jeder internen Nervenzelle findet ein Integrationsprozess der jeweiligen anliegenden Ströme aus Stimulus ($I_{Stimuli}$), anderen chemikalischen Synapsen (I_{Syn}) und Gap-Junctions (I_{Gap}) statt. Durch das Leaky Integrate and Fire - Modell kann das Membranpotential durch anliegende Ströme zum nächstgelegenen Zeitpunkt bestimmt und ein mögliches Feuer-Event vorhergesagt werden. Eine Nervenzelle feuert ein Signal, wenn das Membranpotential einen Threshold $\theta = -20mV$ erreicht hat. Neurotransmitter werden freigelassen und ein Informationsfluss findet statt.

Um nun den Reflex des Wurms C. Elegans umzusetzen benötigt es noch zwei *Motor-Neuronen*. Diese sind dafür zuständig, ein Befehl in Form eines Feuer-Signals an gewisse Muskelgruppen zu übersetzen, damit diese bewegt werden. In dem behandelten Experiment bedient die Inter-Neurone AVA die Motor-Neurone REV, welche für eine Rückwärtsbewegung steht, analog die Inter-Neurone AVB die Motor-Neurone FWD, welche eine Vorwärtsbewegung initiiert.

Dieser Kreislauf bildet nun ein in sich geschlossenes System mit vier Eingängen und zwei Ausgängen (man achte auf das Mapping mit positiven und negativen Werten) und bildet ein lernfähiges neuronales Netz.

1.4 Das symmetrische neuronale Netz

Wie in [6] bereits thematisiert, wird in Abb. 1.3 lediglich eine Hälfte des symmetrischen neuronalen Netzes des Wurms C. Elegans beschrieben. Wie im menschlichen Gehirn besteht das Netzwerk aus zwei Hälften, welche zusammenwirken und bei gegebenen Sensor-Input eine Aktion wählen. Eine erweiterte Analyse des Netzwerks besonders mit den berechneten Gewichten der einzelnen Synapsen ergibt, dass das gegebene Netz von Lechner et al. 1.3 unsymmetrisch scheint. Die Nervenzelle DVA, welche als Synchronisationszelle zwischen beiden Netzwerkhälften

dienen soll, taucht im gegebenen Netz als unsymmetrische Komponente auf und scheint gewisse Sensor-Inputs ungleichmäßig zu gewichten. Im Zuge dessen wird ein neues, symmetrisches neuronales Netz entwickelt, welches zum Einen symmetrischer Natur ist, zum Anderen manche Synapsen und Gap-Junctions misst, da diese nicht zielführend für das gegebene Problem erschienen. Spätere Simulationen bestätigten diese Annahmen, indem durch Gewichtung der Synapsen und Gap-Junctions manche Verbindungen ein verschwindend geringes Gewicht erhielten.

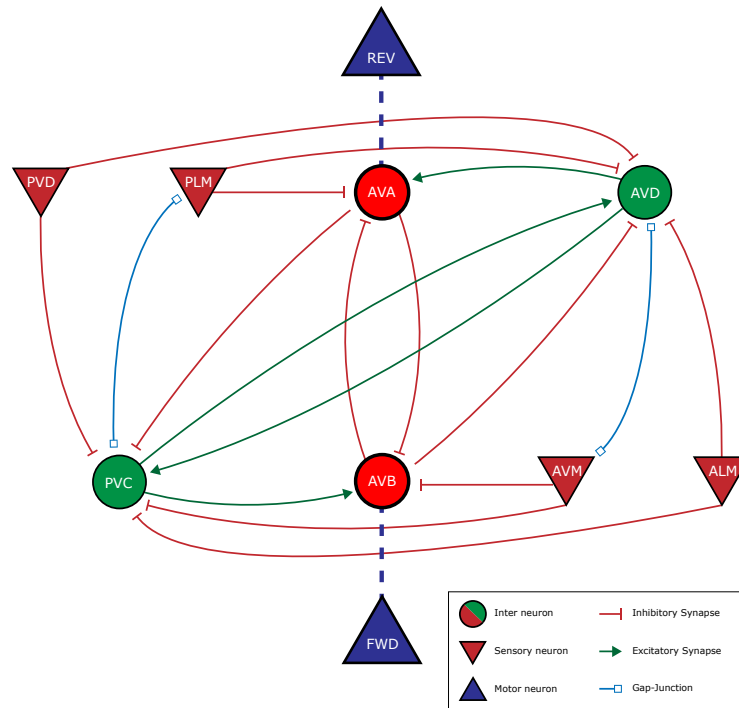


Abb. 1.4: Symmetrisches neuronales Netz des TW-Circuits

Das in Abb. 1.4 dargestellte, symmetrische neuronale Netz des TW-Circuit wird für alle weiteren Analysen und Simulationsläufe verwendet. Die genaue Umsetzung wird in Kapitel 4 weiter erläutert.

Kapitel 2

Leaky Integrate and Fire und simulative Modelle neuronaler Netze

Um biologische neuronale Netze zu simulieren und nutzbar zu machen, bedarf es verschiedener Modelle und Algorithmen. Dieses Kapitel stellt das Leaky Integrate and Fire - Modell vor, welches zur Berechnung des Membranpotentials einer internen Nervenzelle dient. Da es sich hier um eine lineare Differentialgleichung erster Ordnung handelt, werden darüber hinaus numerische Berechnungsmethoden vorgestellt, welche ebenfalls implementiert werden. Weiterhin wird auf die Berechnung der Synapsenströme und Übersetzung der Sensorpotentiale eingegangen und ein simulatives Modell des neuronalen Netzes vorgestellt.

2.1 Das Leaky Integrate and Fire - Modell

Grundsätzlich wird in der Natur beobachtet, dass die neuronale Dynamik als Summationsprozess gefolgt von einer kurzfristigen Entladung des Aktionspotentials beschrieben werden kann. Die Entladung erfolgt hierbei immer ab einem gewissen Wert, welcher als 'Threshold' ϑ beschrieben wird. Bei Überschreitung 'feuert' die Nervenzelle und Informationen gelangen über Synapsen und Gap-Junctions zu nahegelegenen Neuronen.

Um dieses Verhalten zu modellieren, wird der Zellkern genauer betrachtet. Dieser ist mit einer Zellmembran umgeben, welche als guter Isolator dient. Bei anliegenden Strömen $I_{Stimuli}$, I_{Syn} oder I_{Gap} wird die elektrische Ladung $q = \int I(t')dt'$ die Membran aufladen. Die Zellmembran handelt entsprechend einem Kondensator mit Kapazität C_m . Da jedoch in der Natur kein perfekter Kondensator existiert, verliert die Zellmembran über Zeit minimal elektrische Ladung. Daher wird dem Kondensator ein Leckwiderstand R parallel geschaltet. Um das beobachtete Ruhepotential U_{Leak} nach einem Feuer-Event oder bei keinem Input wiederherzustellen, wird eine Batterie in Reihe mit dem Widerstand R geschaltet.

Technisch lässt sich dieses Verhalten als ein elektrisches Ersatzschaltbild wie in Abb. 2.1 darstellen (siehe [8] Kap. 1.3.1). Um nun eine geeignete Differentialgleichung herzuleiten, wird zuerst das erste Kirchhoffsche Gesetz angewendet

$$I(t) = I_R + I_C. \quad (2.1)$$

Der erste Strom I_R ist einfach durch das ohmsche Gesetz wie folgt zu berechnen

$$I_R = \frac{U_R}{R} = \frac{U - U_{Leak}}{R}. \quad (2.2)$$

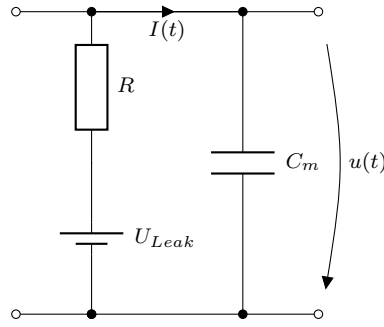


Abb. 2.1: Ersatzschaltbild der Zellmembran

Der Strom I_C wird durch die Definition eines Kondensators $C = \frac{q}{U}$ zum kapazitiven Strom

$$I_C = \frac{dq}{dt} = C_m \frac{dU}{dt}. \quad (2.3)$$

Hierbei steht q für die elektrische Ladung und U für die anliegende Spannung.

Gleichungen 2.2 und 2.3 eingesetzt in Gleichung 2.7 ergibt

$$I(t) = \frac{u(t) - U_{Leak}}{R} + C \frac{du}{dt}. \quad (2.4)$$

Wird diese Gleichung mit R multipliziert und etwas umgestellt, bildet sich die folgende lineare Differentialgleichung erster Ordnung:

$$RC \frac{du(t)}{dt} = (U_{Leak} - u(t)) + RI(t). \quad (2.5)$$

Nach Division durch RC und Einführung des Leitwerts $G_{Leak} = \frac{1}{R}$ entsteht unsere gewollte Form:

$$\frac{du(t)}{dt} = \frac{G_{Leak}(U_{Leak} - u(t)) + \sum_{i=1}^n I_{in}}{C_m}. \quad (2.6)$$

In dieser Gleichung stehen die Variablen G_{Leak} , U_{Leak} und C_m für Parameter der betrachteten Nervenzelle, während $I(t) = \sum_{i=1}^n I_{in}$ stellvertretend für alle eingehenden Ströme aus Stimuli, chemischen Synapsen und Gap-Junctions steht

$$I_{in} = I_{Stimuli} + I_{Syn} + I_{Gap}. \quad (2.7)$$

Die Implementierung dieser Gleichungen und den entsprechenden numerischen Lösungsverfahren findet sich in ???. Ein beispielhafter Spannungsverlauf bei einem konstanten, positiv einfließendem Strom I_{in} sieht wie folgt aus:

Die anliegenden Synapsenströme sind durch folgenden formelaren Zusammenhang zu berechnen:

$$I_{Syn} = \frac{w}{1 + e^{\sigma(u_{pre}(t) + \mu)}} (E - u_{post}(t)). \quad (2.8)$$

Synapsenströme sind grundsätzlich von den pre- und postsynaptischen Potentialen der jeweiligen Nervenzellen u_{pre} und u_{post} abhängig. Weiterhin können diese chemischen Synapsen exzitatorisch oder inhibitorisch wirken. Diese Eigenschaft wird durch das s.g. Nernstpotential $E \in [0mV, -90mV]$ beschrieben. Weitere Größen dieser Gleichung bilden w , die Standardabweichung σ und μ .

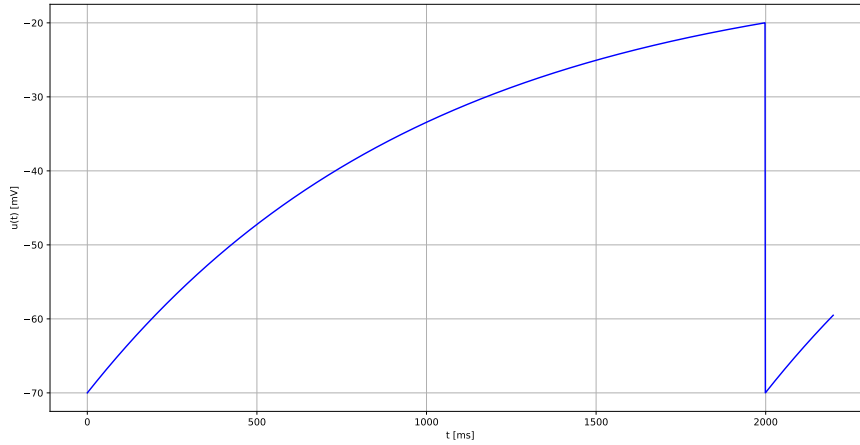


Abb. 2.2: Graphische Darstellung des Membranpotentials durch das Leaky Integrate and Fire - Modell

Gap-Junctions bilden die Ausnahme, denn sie dienen als Ausgleichsglied und wirken bidirektional. Ihr Strom wird wie folgt berechnet:

$$I_{Gap} = \hat{w}(u_{post}(t) - u_{pre}(t)). \quad (2.9)$$

Für die Berechnung des Gap-Junction Stroms benötigt es ebenfalls das pre und postsynaptische Potential der jeweiligen Nervenzellen u_{pre} und u_{post} , sowie \hat{w} .

Durch diese formalen Zusammenhänge kann ein ganzheitliches neuronales Netz simuliert werden.

2.2 Anwendung auf Modelle neuronaler Netze

Durch das im vorherigen Kapitel beschriebene Leaky Integrate and Fire - Modell ist es möglich, interne Vorgänge eines neuronalen Netzes zu beschreiben und zu simulieren. Dies setzt jedoch einen konstanten Input der vier Sensorneuronen durch äußere Stimuli voraus. Diese Rezeptorneuronen sind in der Lage, äußere Einflüsse wie bspw. Licht- oder Berührungsintensität in ein für das neuronale Netz verständliche Größe zu übersetzen. Wie bereits eingangs erwähnt, bewegen wir uns in einem Aktionsraum $A \in [-70mV, -20mV]$, wobei $-70mV$ als Ruhespannung und $-20mV$ als Aktionspotential wahrgenommen wird. Aufgabe der vier Sensor-Neuronen *PVD*, *PLM*, *AVM* und *ALM* ist es folglich, eingehende Größen entsprechend auf den gegebenen Aktionsraum A zu übersetzen.

In dem bereits thematisierten Schaubild nach Lechner et al (Abb. 1.3 [3]) werden jeweils zwei Sensorneuronen für einen Eingang genutzt, da zwischen positiven und negativen Eingangsgrößen unterschieden wird. *PLM* und *AVM* bilden das primäre Sensorpaar für die ausschlaggebendste Eingangsgröße (inverses Pendel: Winkel φ), *PVD* und *ALM* bedienen eine sekundäre Eingangsgröße (inverses Pendel: Winkelgeschwindigkeit $\dot{\varphi}$ oder Kartposition x). Diese Wahl beruht auf der internen Verschaltung des Netzwerks durch Synapsen und Gap-Junctions und wird im weiteren Verlauf dieser Arbeit weiter thematisiert.

Um nun die jeweiligen Größen durch die Sensorneuronen zu übersetzen werden folgende Funktionen für die jeweils positive und negative Sensorneurone $S_{positiv}$ und $S_{negativ}$ angenommen:

$$S_{positiv} := \begin{cases} -70mV & x \leq 0 \\ -70mV + \frac{50mV}{x_{min}}x & 0 < x \leq x_{min} \\ -20mV & x > x_{max} \end{cases} \quad (2.10)$$

$$S_{negativ} := \begin{cases} -70mV & x \geq 0 \\ -70mV + \frac{50mV}{x_{min}}x & 0 > x \geq x_{min} \\ -20mV & x < x_{max} \end{cases} \quad (2.11)$$

$x \in [x_{min}, x_{max}]$ ist eine messbare, dynamische Systemvariable, welche in den gegebenen Grenzen x_{min} und x_{max} auftritt. Lediglich eine Fallunterscheidung wird getroffen: nimmt x einen positiven Wert an, wird Sensorneurone $S_{positiv}$ aktiviert, bei negativem x -Wert, agiert die Sensorneurone $S_{negativ}$.

Analog lässt sich dieser Zusammenhang auf die beiden Motorneuronen REV und FWD übertragen. Hier werden die Signale der internen Nervenzellen AVA und AVB auf interpretierbare Größen in die Außenwelt übersetzt. Biologisch kann dies ein Nervenimpuls sein, welcher eine spezielle Muskelgruppe anspricht oder einen Reflex auslöst. In der hier genannten Simulationsumgebung des inversen Pendels entspricht der Ausgang des Netzwerks entweder einer diskreten Vorwärts- oder Rückwärtsbewegung. Genauer zu der Interaktion mit dem genannten Simulationskonstrukt im Kapitel 4.

2.3 Zuverlässigkeit und Limitationen

Das Leaky Integrate and Fire - Modell ist stark vereinfacht und zeigt die grundsätzlichen Eigenschaften des Membranpotentials auf. Es erfolgt ein lineares Aufintegrieren der anliegenden Ströme und eine simple Rücksetzung des Aktionspotentials nach Überschreitung des Thresholds ϑ auf das Ruhepotential U_{Leak} .

Zur weiteren Analyse eines neuronalen Netzwerks besonders im Bereich der Biologie und Biochemie werden daher detailliertere Modelle angewendet um biologische Effekte in verschiedenen Zelltypen zu berücksichtigen. Jedoch eignet sich das hier angewendete Modell sehr gut zur Analyse der gegebenen Nervenzellen. Das Leaky Integrate and Fire - Modell ist in der Lage, s.g. Fire-Events bei der Überschreitung des genannten Thresholds exakt zu ermitteln und liefert somit eine grundlegende Zeitbasis für die Simulationsumgebung.

2.4 Implementierung

Zur Implementierung des Leaky Integrate and Fire - Modells wird die Programmiersprache `Python` verwendet. Angelehnt an die Formeln aus 2.1 kann ein einfacher Algorithmus implementiert werden. Der gesamte Code findet sich in Anhang A.1.

Da sich in der Berechnung der Membranpotentiale eine lineare Differentialgleichung erster Ordnung ergibt 2.6, muss diese entsprechend numerisch gelöst werden. Die Lösung kann durch das

Euler-Verfahren, sowie durch die Methode nach Runge-Kutta gefunden werden, wobei letztere Methode (4. Ordnung) deutlich genauer ist.

Anmerkung 2.1 (Numerisches Lösungsverfahren nach Euler).

Gegeben sei eine Differentialgleichung der Form $\dot{x} = f(x)$ mit der Bedingung $x = x_0$ bei $t = t_0$. Man finde einen Weg, um die Lösung $x(t)$ zu approximieren.

Weiterhin sollte die Schrittweite Δt bekannt sein sowie die Anzahl der Zeitschritte T . Somit lässt sich die Differentialgleichung numerisch lösen:

$$x_{n+1} = x_n + f(x_n)\Delta t \quad (2.12)$$

Aus [5].

Anmerkung 2.2 (Numerisches Lösungsverfahren nach erweiterter Euler-Methode).

Gegeben sei ebenfalls eine Differentialgleichung der Form $\dot{x} = f(x)$ mit der Bedingung $x = x_0$ bei $t = t_0$. Man finde einen Weg, um die Lösung $x(t)$ zu approximieren.

Weiterhin sollte die Schrittweite Δt bekannt sein sowie die Anzahl der Zeitschritte T . Somit lässt sich die Differentialgleichung numerisch lösen:

$$\tilde{x}_{n+1} = x_n + f(x_n)\Delta t \quad (2.13)$$

$$x_{n+1} = x_n + \frac{1}{2}[f(x_n) + f(\tilde{x}_{n+1})]\Delta t \quad (2.14)$$

Dieses Verfahren ermöglicht eine genauere Approximation als die einfache Euler-Methode bei gleichbleibender Schrittweite. Der Fehler $E = |x(t_n) - x_n|$ wird kleiner. Aus [5].

Anmerkung 2.3 (Numerisches Lösungsverfahren nach Runge-Kutta 4. Ordnung).

Gegeben sei eine Differentialgleichung der Form $\dot{x} = f(x)$ mit der Bedingung $x = x_0$ bei $t = t_0$. Man finde einen Weg, um die Lösung $x(t)$ zu approximieren.

Weiterhin sollte die Schrittweite Δt bekannt sein sowie die Anzahl der Zeitschritte T . Somit lässt sich die Differentialgleichung numerisch lösen:

$$\begin{aligned} k_1 &= f(x_n)\Delta t \\ k_2 &= f(x_n + \frac{1}{2}k_1)\Delta t \\ k_3 &= f(x_n + \frac{1}{2}k_2)\Delta t \\ k_4 &= f(x_n + k_3)\Delta t \end{aligned} \quad (2.15)$$

$$x_{n+1} = x_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \quad (2.16)$$

Dieses Verfahren ermöglicht eine genauere Approximation als die Euler-Methoden bei gleichbleibender Schrittweite. Der Fehler $E = |x(t_n) - x_n|$ wird signifikant kleiner. Diese Methode erfordert jedoch eine höhere Rechenzeit und ist daher nur bei ausreichender Leistung anzuwenden. Aus [5].

Anwendung der Anmerkung 2.3 auf die Funktion 2.6 resultiert in folgende Berechnung, welche direkt implementiert werden kann:

$$\begin{aligned}
k_1 &= \frac{G_{leak}(U_{leak} - u_i(t)) + (I_{stimuli} + I_{syn} + I_{gap}) \Delta t}{C_m} \\
k_2 &= \frac{G_{leak}(U_{leak} - (u_i(t) + \frac{1}{2}k_1)) + (I_{stimuli} + I_{syn} + I_{gap}) \Delta t}{C_m} \\
k_3 &= \frac{G_{leak}(U_{leak} - (u_i(t) + \frac{1}{2}k_2)) + (I_{stimuli} + I_{syn} + I_{gap}) \Delta t}{C_m} \\
k_4 &= \frac{G_{leak}(U_{leak} - (u_i(t) + k_3)) + (I_{stimuli} + I_{syn} + I_{gap}) \Delta t}{C_m}
\end{aligned} \tag{2.17}$$

Rekursive Berechnung der vier Koeffizienten führt zum neuen Membranpotential und entsprechend zu der Information, ob die internen Nervenzellen *AVA* oder *AVB* gefeuert haben:

$$u_{i+1}(t) = u_i(t) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{2.18}$$

Um diese Funktion im Gesamtcode später mühelos aufrufen zu können, wird ein Python-Script (`modules/lif.py`) erstellt. Dieses enthält neben der Funktion zur Berechnung des Membranpotentials auch die der Berechnung von Synapsen- und Gap-Junction-Strömen.

In diversen Such-Algorithmen wird eine Funktion `compute` implementiert, welche das Modul `lif.py` aufruft. Um eine effiziente Berechnung der Membranpotentiale und Synapsen- bzw. Gap-Junction Ströme zu gewährleisten, wird die Funktion wie folgt aufgebaut: Die Vektoren \mathbf{x}

Algorithmus 1: compute

```

Input :  $\mathbf{x}, \mathbf{u}, \mathbf{A}, \mathbf{B}, C_m, G_{Leak}, U_{Leak}, \sigma, \mathbf{w}, \hat{\mathbf{w}}$ 
Output:  $\mathbf{x}, \mathbf{u}, \mathbf{fire}, I_{syn}, I_{gap}$ 
1 for  $i \leftarrow 1$  to 4 do
2   for  $j \leftarrow 1$  to 4 do
3     if  $A_{i,j} == 1$  then
4        $I_{inter_{i,j}} = I_{syn\_calc}(x[i], x[j], E_{in}, w[k], \sigma[k], \mu)$ 
5        $k \leftarrow k + 1$ 
6     else if  $A_{i,j} == 2$  then
7        $I_{inter_{i,j}} = I_{syn\_calc}(x[i], x[j], E_{ex}, w[k], \sigma[k], \mu)$ 
8        $k \leftarrow k + 1$ 
9     else
10       $I_{inter_{i,j}} = 0$ 
11    end
12    if  $B_{i,j} == 1$  then
13       $I_{sensor_{i,j}} = I_{syn\_calc}(u[i], u[j], E_{in}, w[k], \sigma[k], \mu)$ 
14       $l \leftarrow l + 1$ 
15    else if  $B_{i,j} == 3$  then
16       $I_{sensor_{i,j}} = I_{gap\_calc}(u[i], x[j], \hat{w}[k])$ 
17       $m \leftarrow m + 1$ 
18    else
19       $I_{sensor_{i,j}} = 0$ 
20    end
21  end
22 end
23 for  $i \leftarrow 0$  to 4 do
24    $I_{inter} = I_{inter}.sum(axis = 0)$ 
25    $I_{sensor} = I_{sensor}.sum(axis = 0)$ 
26    $x[i], \mathbf{fire}[i] = U_{neuron}.calc(x[i], I_{inter}[i], I_{sensor}[i], C_m[0, i], G_{Leak}[0, i], U_{Leak}[0, i], v, \Delta t)$ 
27 end
28 return  $\mathbf{x}, \mathbf{u}, \mathbf{fire}, I_{syn}, I_{gap}$ 

```

und \mathbf{u} spiegeln die aktuellen Membranpotentiale der jeweiligen Nervenzellen wieder:

$$\mathbf{x} = (AVA \ AVD \ PVC \ AVB) \text{ und} \quad (2.19)$$

$$\mathbf{u} = (PVD \ PLM \ AVM \ ALM). \quad (2.20)$$

Vektor \mathbf{x} beschreibt das Membranpotential interner Nervenzellen, Feuer-Events der Neuronen AVA und AVB sind später von Interesse. \mathbf{u} stellt das Potential für die Eingangsneuronen dar, welche durch die Berechnungsvorschriften 2.10 und 2.11 je nach Sensordaten gesetzt werden. Matrizen \mathbf{A} und \mathbf{B} werden als Transitionsmatrizen genutzt, um die Verbindungen der Neuronen im neuronalen Netz zu beschreiben.

$$\mathbf{A} = \begin{array}{c} \begin{array}{c} \uparrow \\ AVA \\ AVD \\ PVC \\ AVB \end{array} \begin{array}{ccccc} & AVA & AVD & PVC & AVB \\ \left(\begin{array}{cccc} 0 & 0 & 1 & 1 \\ 2 & 0 & 2 & 0 \\ 0 & 2 & 0 & 2 \\ 1 & 1 & 0 & 0 \end{array} \right) \end{array} \quad (2.21)$$

beschreibt Verbindungen innerhalb der internen Nervenzellen. Dabei stehen die Ziffern $[0, 1, 2]$ für folgende Verbindungen:

- 0 : Keine Verbindung zwischen Neuronen \mathbf{A}_i und \mathbf{A}_j
- 1 : Inhibitorische Verbindung zwischen Neuronen \mathbf{A}_i und \mathbf{A}_j
- 2 : Exzitatorische Verbindung zwischen Neuronen \mathbf{A}_i und \mathbf{A}_j .

Die Matrix

$$\mathbf{B} = \begin{array}{c} \begin{array}{c} \uparrow \\ PVD \\ PLM \\ AVM \\ ALM \end{array} \begin{array}{ccccc} & AVA & AVD & PVC & AVB \\ \left(\begin{array}{cccc} 0 & 1 & 1 & 0 \\ 1 & 1 & 3 & 0 \\ 0 & 3 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{array} \right) \end{array}. \quad (2.22)$$

beschreibt analog die Verbindungen der Sensor-Neuronen mit den internen Nervenzellen. Verbindungen werden zwischen diesen Typen von Nervenzellen nur durch inhibitorische Synapsen und Gap-Junctions gemacht. Die Ziffern $[0, 1, 3]$ haben folgende Bedeutung:

- 0 : Keine Verbindung zwischen Neuronen \mathbf{B}_i und \mathbf{B}_j
- 1 : Inhibitorische Verbindung zwischen Neuronen \mathbf{B}_i und \mathbf{B}_j
- 3 : Verbindung durch Gap-Junction zwischen Neuronen \mathbf{B}_i und \mathbf{B}_j .

Kapitel 3

Reinforcement Learning - Lernen mit Belohnung

Reinforcement Learning (kurz: RL) kann als einer der drei großen Bereiche des Maschine Learning interpretiert werden. Neben den Bereichen „Supervised-“ und „Unsupervised Learning“ deckt es ein weites Spektrum an Anwendungsfeldern ab.

Die grundsätzliche Vorgehensweise im Reinforcement Learning ist simpel: Ein Agent ist in der Lage, eine Simulation oder ein Spiel zu bedienen. Seine Aktion beeinflusst eine gut bekannte Umwelt bzw. Simulationsumgebung. Die Ergebnisse dieser Aktion werden durch das Beobachten der Umwelt bzw. der Simulation interpretiert und eingeschätzt. Der Lernprozess erfolgt, indem der Agent durch die Interpretierung der Observation und einer Belohnung eine Aktion tätigt, welche die Belohnung maximieren soll. Durch die immer größer werdende Datenbasis fällt es dem Agenten mit fortgeschrittener Simulation immer einfacher, die „richtigen“ Aktionen zu treffen, um die maximale Belohnung zu erhalten.

3.1 Reinforcement Learning - eine Abwandlung des Deep Learning

Deep Learning hat in den letzten Jahren immer mehr an Relevanz gewonnen. Obwohl der Grundstein dieser Algorithmen und Vorgehensweisen bereits Ende des 19. Jahrhunderts gelegt wurde, fehlte es damals an Rechenleistung sowie hoch parallelen Rechenstrukturen. In der Theorie ist das Konstrukt des Deep Learning in der Lage, bei gegebenen Berechnungsmodellen mit multiplen verbundenen Ebenen Strukturen in großen Datenmengen zu erkennen. Durch heutige Rechenleistungen können Strukturen ein beliebig hohes Abstraktionslevel aufweisen. Anwendungsbereiche für Deep Learning bewegen sich meist im Bereich der Bild- oder Spracherkennung und -klassifizierung, breiten sich jedoch auch auf weitere Bereiche wie Medizin (Pharmazie, Genom-Entschlüsselung) oder Wirtschaft (Kunden-Kaufverhalten, Logistik) aus. Dabei zeichnet einen guten Deep Learning Algorithmus die Fähigkeit aus, s.g. Raw-Files (unbearbeitete Signale wie bspw. Audio-Dateien oder Bilder) ohne Vorwissen auf die gewünschten Daten zu untersuchen und zu klassifizieren, ohne aufwändige Filter, Feature-Vektoren oder andere Mittel zur Vorklassifikation.

Supervised Learning (zu Deutsch: überwachtes Lernen) bildet die Grundlage und wurde in den Anfängen der künstlichen Intelligenz eingesetzt. Ein Algorithmus lernt aus gegebenen Paaren von Ein- und Ausgängen eine Funktion, welche nach mehrmaligen Trainingsläufen Assoziationen herstellen soll und auf neue Eingaben passende Ausgaben liefert.

Unsupervised Learning (zu Deutsch: unüberwachtes Lernen) bietet entgegen der Methode des

supervised Learning die Möglichkeit, ein Modell ohne im Voraus bekannte Zielwerte oder Belohnungssysteme durch die Umwelt zu trainieren. Entsprechend benötigen diese Algorithmen mehr Rechenleistung (bei gleichbleibender Aufgabenstellung). Sie versuchen, in einer Anhäufung von Daten Strukturen zu erkennen, welche von stochastischem Rauschen abweichen. Neuronale Netze orientieren sich hier oft an den bekannten Eingängen. Diese Methode wird oft in Bereichen der automatischen Klassifizierung oder Dateikomprimierung genutzt, da hier das Ergebnis im Vorhinein meist unbekannt ist.

Reinforcement Learning bietet, wie bereits in der Einleitung erwähnt, den Vorteil eines Reward-Systems (zu Deutsch: Belohnungssystem).

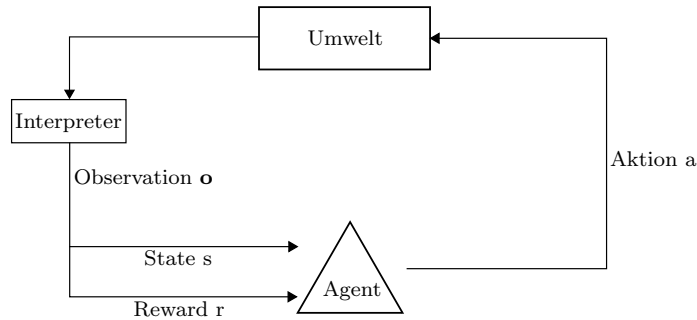


Abb. 3.1: Graphische Darstellung des Reinforcement Learning Algorithmus

Der Agent beginnt mit einer anfangs willkürlich gewählten Aktion und beeinflusst damit die Umwelt bzw. die Simulation. Durch einen Interpreter ist es möglich, wichtige Messgrößen (inverses Pendel: Winkel φ oder Winkelgeschwindigkeit $\dot{\varphi}$) zu messen und in einen Observationsvektor \mathbf{o} zu schreiben. Dieser kann ausgelesen werden und den aktuellen State x_i nach der erfolgten Aktion liefern. Dazu wird durch ein anfangs definiertes Reward-System ein vereinbarter Reward geliefert, welcher die Performance der Simulation widerspiegelt. Der Agent besitzt nun diese Informationen und entscheidet aufgrund des gegebenen States sowie des Rewards, welche Aktion als nächstes getätigt werden soll. In der Theorie wird so der Reward mit jeder erfolgreichen Episode höher und der Agent ist in der Lage gewisse Parameter der Simulation entsprechend des jeweiligen Observationsparameters anzupassen.

Bei dieser Methode ist die Grundlage aller Algorithmen und Optimierungsverfahren der Gesamterward

$$G_t = \sum_{k=0}^T R_{t+k+1} \quad (3.1)$$

Des weiteren ist es geläufig, einen s.g. „Discount-Faktor“ γ einzuführen. Rewards in frühen Schritten der Simulation sind wahrscheinlicher und gut vorherzusehen, wohingegen in fortgeschrittenen Simulationen die Aktionen meist schwer vorhersehbar sind und somit einen höheren Reward verdienen.

$$G_{t\gamma} = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \text{ mit } \gamma \in [0, 1) \quad (3.2)$$

Von der Benutzung eines solchen Discount-Faktors wird jedoch zuerst abgesehen, da das Finden der perfekten Parameter für die vorgestellte Simulation im Vordergrund steht. In weiteren Anwendungen kann dieser Faktor eingeführt werden.

3.2 Anwendung auf Modelle neuronaler Netze

Reinforcement Learning findet klassischerweise Anwendung durch Deep Learning Algorithmen auf künstlich erstellte neuronale Netze mit vielen s.g. "Hidden Layers,, (Ebenen zwischen Ein- und Ausgang mit hoher Anzahl an Neuronen) statt. Variiert werden in einem solchen Netz lediglich die jeweiligen Gewichte der Synapsen zwischen den Neuronen. Synapsen sind darüber hinaus einfache Mittel zur Informationsübertragung und haben keine weiteren Eigenschaften oder zeigen kein eigenes Verhalten.

Das hier vorliegende neuronale Netzwerk ist jedoch gänzlich anders aufgebaut. Nervenzellen werden durch Potentiale beschrieben und integrieren anliegende Informationen auf. Synapsen können verschiedenen Typs sein und hemmend sowie erregend wirken. Sowohl Nervenzellen als auch Synapsen (und Gap-Junctions) haben verschiedene Parameter, welche gewisse Aktionen im neuronalen Netz verursachen können.

Daher wird in dieser Arbeit die Methode des Reinforcement Learning auf biologische neuronale Netze abgewandelt, um dieses auf Probleme der Regelungstechnik anzuwenden. Folglich befassen wir uns mit einem neuronalen Netz, welches eine Ebene mit vier Neuronen aufweist. Diese Neuronen arbeiten wie bereits in Kapitel 1 und 2 beschrieben ebenfalls anders als in den üblichen Modellen künstlicher neuronaler Netze.

Die Schwierigkeit dieser Aufgabenstellung besteht darin, geeignete Parameter für jede Nervenzelle sowie für jede Synapse zu finden, sodass das Netz korrekt und zuverlässig auf interpretierte Signale aus der Umwelt reagiert und entsprechend durch den Agenten eine Aktion wählt, welche einen möglichst hohen Reward nach sich zieht. Bezogen auf Abb. 3.1 stellt die Umwelt unsere Simulationsumgebung des inversen Pendels (`OpenAI Gym - CartPoleV0`) dar. Diese nimmt eine Aktion (FWD oder REV) pro Simulationsschritt an und gibt entsprechend einen Observationsvektor \mathbf{o} aus, welcher durch den Interpreter übersetzt wird. Der aktuelle State s wird durch die vier Sensorneuronen *PVD*, *PLM*, *AVM* und *ALM* entsprechend interpretiert und in das neuronale Netz eingegeben. Durch den gegebenen Reward haben wir die Information, wie gut das Netzwerk in der entsprechenden Episode mit den entsprechenden Parametern abschneidet.

3.3 Verschiedene Suchalgorithmen

Die Wahl des geeigneten Suchalgorithmus ist immer von der Beschaffenheit der Problemstellung abhängig. Klassische Probleme mit Anwendung des Reinforcement Learning auf künstliche neuronale Netze nutzen Algorithmen wie Q-Learning, Policies (Epsilon-Greedy, Gradient-Decend/Acend) oder genetische Algorithmen. Diese sind hoch spezialisiert und suchen nach Maxima/Minima der gegebenen Funktion, um den Fehler zu reduzieren. Bei einer großen Zahl an Parametern, welche untereinander noch korreliert sein können, kommt oft der einfache, jedoch gleichzeitig sehr effektive „RandomSearch“ Algorithmus zum Einsatz.

Grundsätzlich sei noch zu erwähnen, dass konventionelle Such- bzw. Optimierungsalgorithmen innerhalb des Reinforcement Learning ein Markov-Entscheidungsproblem voraussetzen.

3.3.1 Q-Learning

Beschreibung und Grundlagen zu Q-Learning

3.3.2 Gradient Policies

Beschreibung und Grundlagen zu Gradient Policies

3.3.3 Genetische Algorithmen

Beschreibung und Grundlagen zu genetischen Algorithmen

3.3.4 Random Search

Beschreibung und Grundlagen zu Random Search + Anwendung

Wie in der vorherigen Sektion 3.2 bereits kurz beschrieben, werden die jeweiligen Parameter der Synapsen und Nervenzellen gesucht. Dies sind die folgenden:

<i>Parameter-Typ</i>	<i>Parameter</i>	<i>Beschreibung</i>	<i>Grenzen</i>
Membranpotential	C_m	Kapazität der Zellmembran	$[1mF, 1F]$
Membranpotential	G_{Leak}	Leitwert der Zellmembran	$[50mS, 5S]$
Membranpotential	U_{Leak}	Ruhepotential der Zellmembran	$[-90mV, 0mV]$
Synapsenstrom	$E_{Excitatory}$	Weiterleiten des Synapsenstroms	$[0mV]$
Synapsenstrom	$E_{Inhibitory}$	Negieren des Synapsenstroms	$[-90mV]$
Synapsenstrom	μ	...	$[-40mV]$
Synapsenstrom	σ	Standardabweichung (Modell)	$[0.05, 0.5]$
Synapsenstrom	w	...	$[0S, 3S]$
Synapsenstrom	\hat{w}	...	$[0S, 3S]$

Die gegebenen Grenzen folgen aus [3] und [2] und sind durch Calcium und Potassiumengen im Nervensystem des C. Elegans verbunden.

Bei dem in Abb. x gezeigten neuronalen Netz handelt es sich um vier interne Nervenzellen, sowie x inhibitorische Synapsen, x exzitatorische Synapsen und x Gap-Junctions.

Kapitel 4

Implementierung des TW-Netzes

Die Implementierung des gesamten neuronalen Netzes inklusive der Simulationsumgebung erfolgt in der Programmiersprache **Python**. Als Module werden zum einen das bereits vorgestellte Leaky Integrate and Fire - Modell implementiert, zum anderen diverse Algorithmen zur Suche von individuellen Parametern des neuronalen Netzes. Darüber hinaus ist das Programm in der Lage, eine Simulation der gefundenen Parameter durch die Simulationsumgebung **CartPole_v0** von OpenAI Gym zu zeigen und Parameter über die Simulationszeit zu plotten. Die Module sowie diverse Dokumentationen und Informationen sind auf meiner GitHub-Repository¹ [7] zu finden.

4.1 Aufbau des Programms

Ein Ziel dieser Arbeit ist es, neben der Funktionstüchtigkeit des Simulators das entstandene Programm mitsamt allen Modulen und Abhängigkeiten modular und leicht verständlich aufzubauen. Dazu gehört eine gute Dokumentation sowie saubere Versionierung, um Änderungen nachvollziehbar darzustellen.

Folgende Anforderungen werden an das Programm gestellt:

- Es existiert ein zentraler Punkt zum ändern aller nötigen Parameter. Alle weiteren Größen werden durch Formeln und Abfragen erzeugt.

Datei: `parameters.py`

- Es wird ein Modul zur Visualisierung gegebener Parameter oder Gewichte des neuronalen Netzes bereitgestellt. Diese Datei muss leicht verständlich und manipulierbar sein, um eigene Plots zu erstellen und neue Simulationsumgebungen einzubinden.

Datei: `visualize.py`

- Es muss die Möglichkeit bestehen, aus bereits existierenden Suchalgorithmen neue Implementationen zu erstellen und gegebene Funktionen einfach einsetzen zu können

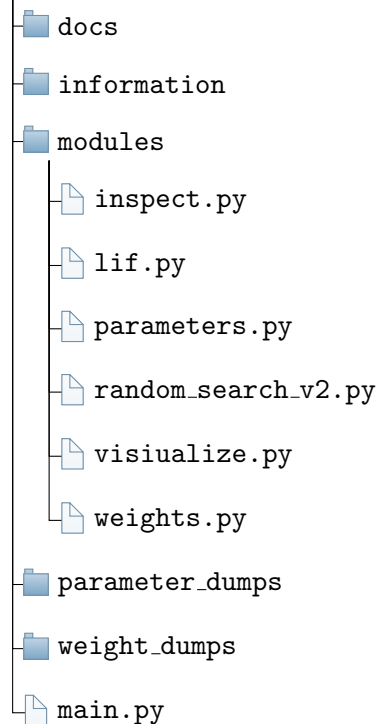
Vorlage: `random_search_v2.py`

¹ <https://github.com/J0nasW/BA>

Aufgrund der hohen Vielfältigkeit an neuronalen Netzen und Suchalgorithmen soll dieses Programm als Basis für neue Projekte und Ideen dienen. Daher empfehle ich stark, eine Fork der Repository [7] zu erstellen und den Simulator weiter zu gestalten.

Das Programm beinhaltet folgende Module:

TW Circuit



- Der Ordner `docs` beinhaltet wichtige Dokumentationen bezüglich des Codes und dem Umgang mit diversen Befehlen innerhalb der `main.py`. Darüber hinaus wird hier ebenfalls diese Arbeit inklusive des \LaTeX -Codes abgelegt.
- Aufgrund vieler komplexer Simulationen mit verschiedenen Parametersätzen wurde die Berechnung von Heim- und Unirechnern auf Rechenzentren ausgelagert. Der Ordner `information` wird genutzt, um Informationen über jede Simulation (Ergebnisse, Zeitstempel, Zugehörigkeit, ...) im Form einer TXT-Datei zu speichern.
- Alle nötigen Module zur Simulation und Visualisierung finden sich in dem Ordner `modules` wieder. Genauere Informationen zu den einzelnen Skripten werden in den nächsten Sektionen aufgeführt.
- `parameter_dumps` und `weight_dumps` sind die Resultate der Simulationsläufe. In diesen Ordnern werden Parameter und Gewichte des neuronalen Netzes nach erfolgreichen Simulationsläufen abgespeichert. Die Dateien werden durch das Skript `hickle` in ein HDF-5 Dateiformat gespeichert.

4.2 Implementierung der Suchalgorithmen

Die Suchalgorithmen befinden sich jeweils in dem Ordner `modules` und werden durch Import in der Datei `main.py` aufgerufen. Sie erhalten bei Aufruf die vom Anwender gewählte Simulationszeit (bspw. 12 Stunden) und bei Bedarf bereits errechnete Parameter. Als Ausgabe wird ein Dump der errechneten Parameter oder Gewichte gespeichert sowie eine Informationsdatei, welche Zugehörigkeiten, Anz. an Simulationen, Laufzeiten und den gesamten Reward beschreibt.

4.2.1 Suchalgorithmus RandomSearch

Der Suchalgorithmus `RandomSearch` wurde direkt in die Simulation eingebunden. Es werden die Parameter $C_m, G_{Leak}, U_{Leak}, \sigma, w, \hat{w}$ durch eine Gleichverteilung in den bereits genannten Grenzen zufällig erzeugt. Um gleichverteilte, zufällige Werte zu generieren, wird die Funktion `random.uniform` aus dem bekannten Package `numpy` verwendet. Nach Aufruf des Algorithmus `random_parameters` wird eine Simulation mit den erzeugten Parametern und maximal 200 Zeitschritten durchgeführt. Der Reward dieser Simulation wird mit vergangenen Rewards verglichen. Wenn die Simulation einen Reward größer oder gleich 200 erreicht, gilt die Simulation als er-

Algorithmus 2: random_parameters

Input : Anz. Nervenzellen, Anz. Synapsen, Anz. Gap-Junctions
Output: Arrays $C_m, G_{Leak}, U_{Leak}, \sigma, w, \hat{w}$
 // Generieren von Zufallsvariablen durch Gleichverteilung.
 // Für Nervenzellen:
 1 $C_m = \text{np.random.uniform}(\text{low} = 0.01, \text{high} = 1, \text{size} = (1, \text{Anz. Nervenzellen}))$
 2 $G_{Leak} = \text{np.random.uniform}(\text{low} = 0.05, \text{high} = 5, \text{size} = (1, \text{Anz. Nervenzellen}))$
 3 $U_{Leak} = \text{np.random.uniform}(\text{low} = -70, \text{high} = 0, \text{size} = (1, \text{Anz. Nervenzellen}))$
 // Für Synapsen:
 4 $\sigma = \text{np.random.uniform}(\text{low} = 0.05, \text{high} = 0.5, \text{size} = (1, \text{Anz. Synapsen}))$
 5 $w = \text{np.random.uniform}(\text{low} = 0, \text{high} = 3, \text{size} = (1, \text{Anz. Synapsen}))$
 6 $\hat{w} = \text{np.random.uniform}(\text{low} = 0, \text{high} = 0.3, \text{size} = (1, \text{Anz. Gap-Junctions}))$ **return** $C_m, G_{Leak}, U_{Leak}, \sigma, w, \hat{w}$

folgreich, andernfalls wird nach Ablauf der Simulationszeit der Algorithmus unterbrochen. Der gesamte Programmablauf gestaltet sich vereinfacht wie folgt: Die elementare Funktion

Algorithmus 3: random_search_v2

Input : Simulationszeit
Output: Simulationsinformation (information.txt), Parameter-Dump als .hkl Datei

```

1 action = episodes = best_reward = 0
2 env = gym.make('CartPole-v0')
3 while True do
4   initialize(Default_U_leak)
5   episodes ← episodes + 1
6    $C_m, G_{Leak}, U_{Leak}, \sigma, w, \hat{w} = \text{random\_parameters}()$ 
7   reward = run_episode( $C_m, G_{Leak}, U_{Leak}, \sigma, w, \hat{w}$ ) ; // Simulation mit neuen Parametern - Siehe Sec.
      4.3
8   if reward ≥ best_reward then
9     Set best_reward ← reward
10    Result = [ $C_m, G_{Leak}, U_{Leak}, \sigma, w, \hat{w}$ ] ; // Für Parameter-Dump
11    if reward ≥ 200 then
12      break; // Exit-Argument, wenn Reward von 200 erreicht wurde
13    end
14  end
15  if elapsed_time > simulation_time then
16    break ; // Exit-Argument, um genaue Laufzeiten zu erzielen
17  end
18 end
19 return information.txt, parameter_dump.hkl, date, best_reward

```

run_episode wird in Sektion 4.3 genauer erläutert. Sie enthält unter anderem auch die Funktion compute, welche in Sektion 2.4 bereits detailliert thematisiert wurde.

4.2.2 Optimierungsalgorithmus Weights

Als erstes Optimierungsverfahren nach erfolgreicher Simulation der Parameter des neuronalen Netzes wird nun der Algorithmus Weights eingesetzt. Dieser lässt die bereits simulierten Parameter importieren und gewichtet jede Synapse mit einem Faktor $g \in [0, 1]$. Durch einfache Simulationen mit willkürlichen Gewichten wird schnell festgestellt, welche Synapsen bei einem Simulationslauf mit hohem Reward eine große Gewichtung erhalten und welche Synapsen nicht förderlich für das Ergebnis sind. Durch diese Simulation war es möglich, das in Kap. 1 vorgestellte symmetrische Neuronale Netz (Abb. 1.4) zu entwickeln. Die dort gezeigten Synapsen erfahren

eine relevante Gewichtung und sind somit wichtig für den Erfolg des neuronalen Netzes. Der Gewichtungsalgorithmus ist rechenintensiver als RandomSearch, da insgesamt 16 Synapsen bzw. Gap-Junctions pro Episode mit zufällig gewählten Gewichten versehen werden, um den Reward zu steigern. Jedoch wird ein im Schnitt um das Dreifache höherer Reward verzeichnet bei gleichbleibenden Parametern, was eine sehr gute Optimierung darstellt. Aufgerufen wird dieser Algorithmus analog zu RandomSearch aus der `main.py`-Datei. Als Input werden die bereits errechneten optimalen Parameter der jeweiligen Nervenzellen und Synapsen sowie die maximale Simulationszeit eingegeben. Ebenfalls gleich dem Algorithmus RandomSearch produziert Weights einen Dump mit den errechneten Gewichten der Synapsen und Gap-Junctions sowie eine Informationsdatei im `.txt`-Format, welche weitere Zugehörigkeitsinformationen sowie die Anzahl an Simulationen und die Dauer enthält.

4.2.3 Simulation in der Google Cloud Platform®

Wie bereits in diesem Abschnitt mehrfach erwähnt, sind die implementierten Suchalgorithmen RandomSearch und Weights äußerst rechenintensiv. Beide Skripte erfordern das zufällige Generieren einer hohen Anzahl an Parametern sowie die Anwendung dieser Parameter auf das gegebene Problem durch numerische Lösungsansätze für Differentialgleichungen (Siehe Sektion 2.4). Es müssen Parameter zwischengespeichert und Dumps auf Festplatten geschrieben werden. Um eine erfolgreiche Simulation des inversen Pendels zu erhalten, muss ein Parametersatz mit hohem Reward gefunden werden, welcher die korrekte Funktionsweise des neuronalen Netzes gewährleistet. Bei dem in Abb. 1.4 gezeigten neuronalen Netz werden die Parameter $C_m, G_{Leak}, U_{Leak}, \sigma, w, \hat{w}$ für Synapsen, Gap-Junctions und Nervenzellen simuliert.

<i>Parameter</i>	<i>Kategorie</i>	<i>Anzahl</i>
C_m	Nervenzelle	4
G_{Leak}	Nervenzelle	4
U_{Leak}	Nervenzelle	4
σ	Synapse	16
w	Synapse	16
\hat{w}	Synapse	2
Gesamt:		46

Somit werden in jedem Simulationslauf zuerst 46 Parameter in gegebenen Grenzen durch eine Gleichverteilung erzeugt und anschließend durch das `compute`-Modul 2.4 die benötigten Synapsenströme und Membranpotentiale errechnet.

Das Modul Weights erzeugt, analog zu RandomSearch zuerst für jede Synapse und Gap-Junction ein gleichverteiltes, zufälliges Gewicht $g \in [0, 1]$.

<i>Parameter</i>	<i>Kategorie</i>	<i>Anzahl</i>
g	Synapse	18
Gesamt:		18

Somit werden insgesamt 18 Gewichte pro Episode erzeugt und auf das Modell angewendet. Zur Berechnung der erforderlichen Ströme und Potentiale wird das `compute`-Modul um die Funktion der Gewichtung erweitert.

Ein solches Simulationsvorhaben wird üblicherweise nicht mehr auf Heimrechnern ausgeführt sondern findet den Weg in die Cloud. Gerade in den letzten Jahren haben Cloud-Computing-Firmen wie Amazon mit AWS, Microsoft mit der Azure Cloud und Google mit der Google Cloud Platform (GCP) an großer Aufmerksamkeit gewonnen. Die einfache Handhabung und Kontrolle über eigene virtuelle Instanzen von ganzen Betriebssystemen erlaubt eine zuverlässige und effiziente Simulation von Parametern. In dieser Angelegenheit wurde sich für die Google Cloud Platform entschieden, da diese ein sehr gutes User Interface hat und kostengünstige, virtuelle Maschinen anbietet. Gemietet wurde ein Server mit dem Standort Frankfurt, welcher über vier virtuelle Intel XEON® Prozessoren sowie 12GB DDR4 Arbeitsspeicher verfügt. Dies erlaubt eine schnelle Simulation von vier Instanzen zur gleichen Zeit sowie den Vorteil, das Langzeitsimulationen von 12 Stunden oder mehr im Hintergrund oder über Nacht erfolgen können.

Auf der virtuellen Instanz wurde ein Linux Ubuntu 18.04 LTS installiert und bereitgestellt. Im nächsten Schritt wird die vorbereitete GitHub Repository [7] auf das System geklont und die benötigten Abhängigkeiten (siehe Anhang) installiert. Durch einen Cronjob werden Skripte ausgeführt und mit `crontab -e` die Simulation zu einer festen Uhrzeit gestartet. Die übliche Simulationszeit beträgt jeweils 12 Stunden.

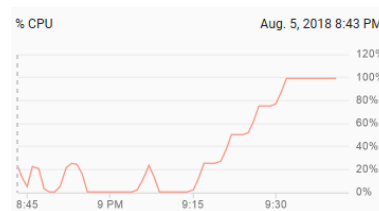


Abb. 4.1: Systemauslastung der virtuellen Instanz auf der GCP

Wie in Abb. 4.1 zu sehen, werden die Suchalgorithmen nacheinander durch den Cronjob ausgeführt, da pro Suchlauf nur ein Prozessorkern in Anspruch genommen werden kann. Der virtuelle Prozessor erreicht somit bei vier gleichzeitigen Simulationsläufen eine Auslastung von 100%. Die Ergebnisse der Suchläufe werden in Form von Parameter- und Weight-Dumps via eines Apache Webservers bereitgestellt und können problemlos auf dem Heimrechner visualisiert werden. Durch das gewählte Dateiformat der Dumps in HDF5 sind die Dateien Plattformunabhängig und äußerst performant lesbar. Diese Methode funktioniert darüber hinaus auch Versionsübergreifend. Eine Simulation kann Dumps mit Python 2.7 erstellen, welche durch einen Heimrechner mit Python 3.6 visualisiert werden können.

4.3 Simulationsumgebung: OpenAI Gym

Um die Performance eigener neuronaler Netze und Algorithmen zu messen, wurden in den letzten Jahren eine ganze Reihe an Simulationen und Spielen entwickelt. Ziel dieser Simulationsumgebungen ist es, die Entscheidung des Agenten in gewissen Situationen zu testen und den Lernerfolg darzustellen. Darüber hinaus wird das Verständnis für neuronale Netze durch das Anwenden auf

Spiele und Experimente vertieft.

Eine sehr bekannte Open Source Bibliothek an Simulatoren und Spielen ist Gym von OpenAI². OpenAI ist eine Non-Profit Organisation mit dem Ziel, den Forschungsbereich der künstliche Intelligenz voran zu treiben und ein besseres Verständnis für die Vorgänge in neuronalen Netzen zu schaffen. Die Bibliothek Gym enthält verschiedene Umgebungen:

- Algorithmen:

Einfache Aktionen wie Copy-Paste, Addition und Subtraktion sowie logische Gatter

- Atari

Sammlung an klassischen Atari Spielen wie Breakout, Pacman und Space Invaders

- Classic Control

Simulationsumgebungen wie das inverse Pendel oder das Mountain Car

- Robotics

Komplexe Simulationen von Roboterarmen oder -händen

In dieser Arbeit wird die Umgebung **CartPole_v0** (Abb. 4.2) gewählt. Diese besteht aus einem einfachen inversen Pendel, welches sich auf einer zweidimensionalen Bahn frei bewegen kann. Die Handhabung dieser Simulationsumgebung ist dank einer großen Community und guten

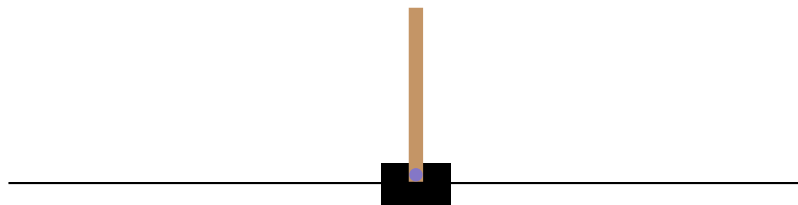


Abb. 4.2: Simulationsumgebung **CartPole_v0**.

Dokumentationen sehr einfach. Initialisiert wird die Umgebung, indem die Bibliothek **gym** in das Python-Skript importiert und die gewählte Umgebung als „Environment“ festgelegt wird. Pro Simulationsschritt kann eine Aktion $a = \{0, 1\}$ getätigt werden.

0 : Schritt nach Links

1 : Schritt nach Rechts

Das bereits vorgestellte Neuronale Netz (Abb. 1.4) verfügt über zwei Motor-Neuronen, welche als Eingang der Simulation genutzt werden können. Interne Nervenzellen *AVA* und *AVB* sind durch eine direkte Verbindung mit den genannten Motor-Neuronen verbunden und verursachen im Falle eines Fire-Events die Bewegung des Wagens um einen Schritt nach links bzw. rechts. Pro Simulationsschritt erfolgt je nach Observation immer ein Fire-Event, sodass das Pendel zu

² <https://openai.com/>

jeder Zeit eine berechnete Aktion erhält.

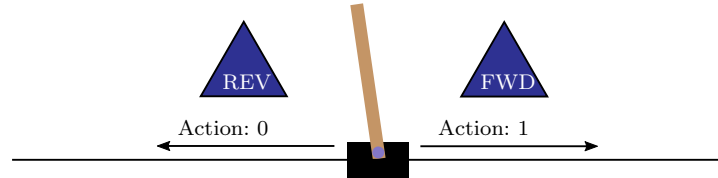


Abb. 4.3: Simulationsumgebung `CartPole-v0` mit Aktionen *FWD* und *REV*.

Pro Simulationsschritt wird ein Observationsvektor ausgegeben. Dieser enthält die folgenden Parameter:

$$\mathbf{o} = (C_{Position} \ C_{Velocity} \ P_{Angle} \ P_{Velocity}). \quad (4.1)$$

Parameter	Beschreibung	Grenzen
$C_{Position}$	Position des Carts	$[-2.4, 2.4]$
$C_{Velocity}$	Geschwindigkeit des Carts	$[-\infty, \infty]$
P_{Angle}	Winkel des Pendels	$[-41, 8^\circ, 41, 8^\circ]$
$P_{Velocity}$	Winkelgeschwindigkeit des Pendels	$[-\infty, \infty]$

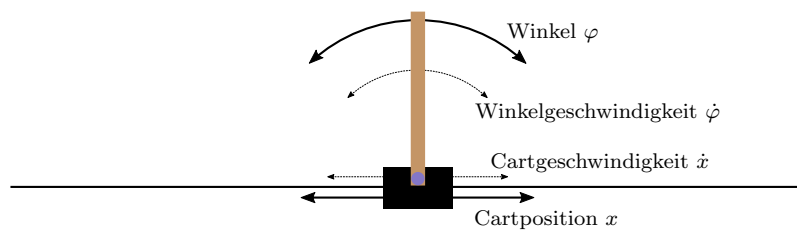


Abb. 4.4: Simulationsumgebung `CartPole-v0` mit Observationsgrößen.

Die Informationen aus dem Observationsvektor \mathbf{o} werden entsprechend interpretiert und den Sensor-Neuronen zugeführt. Wie bereits in Sektion 1.3 beschrieben, ist der Winkel des Pendels φ als primäre Größe den Sensor-Neuronen *PLM* und *AVM* zuzuführen. Als sekundäre Größe kann die Winkelgeschwindigkeit $\dot{\varphi}$ oder die Position des Carts x den Sensor-Neuronen *ALM* und *PVD* zugeführt werden. Die Wahl der geeigneten sekundären Observationsgröße wird in Sektion ? näher erläutert.

4.4 Visualisierung und Auswertung

Wie bereits in Sektion 4.3 näher beschrieben, liefert die Simulationsumgebung `CartPole_v0` eine sehr gute Echtzeitvisualisierung des inversen Pendels als Animation. Diese wird durch das Speichern eines errechneten RGB-Tensors jedes Simulationsschrittes und anschließenden Animieren der Informationen erreicht. Weiterhin soll das Programm jedoch in der Lage sein, wichtige Parameter der Simulation über die Simulationszeit darzustellen, um ggf. Probleme zu erkennen und Aktionen zu verstehen. Um Plots aus Arrays zu erstellen, wird die Python-Bibliothek `matplotlib` genutzt. Durch die einfache Bedienung und den enormen Umfang an Werkzeugen ist diese Bibliothek sehr beliebt in der Visualisierung von Daten und Parametern.

Der generelle Ablauf des Programms sieht im ersten Schritt die Simulation von Parametern und Gewichten vor. Dies erfordert keine Visualisierung der Vorgänge, da die Performance erheblich beeinträchtigt werden würde. Daher wurde das Modul `visualize.py` geschrieben, um gefundene Parameter und Gewichte mühelos simulieren zu können. Für den Aufruf werden die entsprechenden Parameter- und Gewichte-Dumps übergeben sowie die gewünschte Simulationszeit (bspw. 5 Sekunden). Die Simulation wird analog in den Suchalgorithmen jedoch mit festen Parametern ausgeführt. Gewünschte Größen werden pro Simulationsschritt gespeichert und letztlich grafisch dargestellt. Besonders die Übersicht über die Membranpotentiale einzelner Nervenzellen bietet einen sehr guten Überblick über die Vorgänge im neuronalen Netz und die Reaktion auf gegebene Sensor-Daten.

Weiterhin wird standardmäßig neben den Membranpotentialen auch der anliegende Synap-

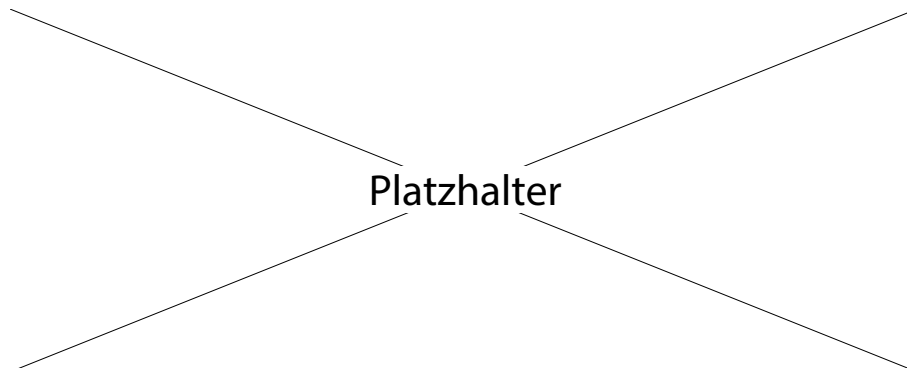


Abb. 4.5: Plot der Membranpotentiale

senstrom an jeder Nervenzelle geplottet. Diese Veranschaulichung gibt Aufschluss über die gewählten Parameter und das Feuerverhalten im internen neuronalen Netzwerk.

4.5 Sonstige Implementierung

Neben den prominenten Modulen wurden im Laufe der Zeit mehrere hilfreiche Nebenmodule und Funktionen implementiert sowie notwendige Pakete genutzt.

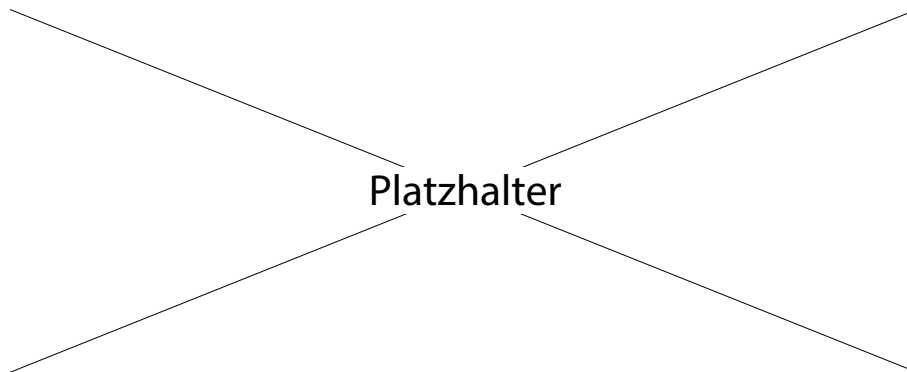


Abb. 4.6: Plot der Membranpotentiale

4.5.1 Zentraler Ort für Parameter

Um einen zentralen Ort für verschiedene Größen und Parameter zu schaffen, wurde das Modul `parameters.py` erstellt. Dieses Modul wird im gesamten Programm eingebunden und dient als globale Informationsbasis. Nur wenige Größen können verändert werden wie bspw. die Transitionsmatrizen des neuronalen Netzes oder manche grundlegende Parameter zur Berechnung von Synapsenströmen und Membranpotentialen. Alle weiteren Daten werden automatisch berechnet und im Laufe der Simulation bereitgestellt.

4.5.2 Speicherung von Daten

Ein Problem, welches über die Implementierung aufkam war die Speicherung von Dateien in s.g. Dump-Files. Nach einer erfolgreichen Simulation von Parametern soll der beste Parametersatz zur weiteren Analyse gespeichert werden. Dazu liefert Python die Bibliothek `Pickle`, welche verschiedene Datentypen seriell in ein kompaktes Binärformat konvertiert und in der `.p` Dateieindung speichert. Dieser Vorgang ist jedoch besonders mit Python 2.7 verhältnismäßig langsam und beeinträchtigt die Performance der Simulation. Darüber hinaus besteht eine Inkompatibilität des Dateiformats zum Einen zwischen verschiedenen Python-Versionen, zum Anderen zwischen verschiedenen Betriebssystemen.

Eine elegantere Möglichkeit bietet die Open Source Bibliothek `hickle`. Sie wird ähnlich wie `pickle` importiert und speichert die gewählten Parametern in einer `.hkl`-Datei. Die Daten werden anders als bei `Pickle` im s.g. HDF5 (Hierarchical Data Format 5) gespeichert. Dies ist zum einen performanter, sorgt zum Anderen für eine erheblich größere Kompatibilität unter Python-Versionen und Betriebssystemen.

4.5.3 Dateiinspektion

Besonders nach den ersten Simulationen ist die Inspektion der errechneten Parametern und Gewichten von äußerster Wichtigkeit. Hohe Rewards bedeuten nicht direkt, dass das neuronale Netz in jeder Situation richtig oder gut performt. Beispielsweise kann ein sehr guter Reward

in der Bewegung des Pendels in Vorwärtsrichtung erreicht werden, obwohl die Parameter der Rückwärtsbewegung nicht sehr gut ausgereift sind. Daher wird ein Modul zur detaillierten Inspektion der Dump-Dateien `inspect.py` geschrieben. Bei Aufruf einer Funktion dieses Moduls muss jeweils der Pfad der Dump-Datei übergeben werden. Die Datei wird geöffnet und die Parameter den entsprechenden Größen zugewiesen. Danach kann ein Konsolenoutput erfolgen oder ein Plot der gespeicherten Daten erzeugt werden.

Kapitel 5

Performance & Auswertung

Ziel dieser Arbeit war es, auf Grundlage eines bereits bestehenden, neuronalen Netzes eine Anwendung des Reinforcement Learning zur Regelung dynamischer Systeme zu schaffen. Das neuronale Netz ist dabei kein konventionell erzeugtes Netz einer künstlichen Intelligenz, sondern beruht auf biologischen Forschungsergebnissen echter Lebensformen.

Somit wurde zuerst eine Berechnungsgrundlage eines solchen Netzes geschaffen und implementiert. Dazu wurden verschiedene numerische Lösungsverfahren von Differentialgleichungen verglichen und umgesetzt. Letztlich wird ein universeller Simulator geschaffen, welcher Informationen über die Nervenzellen und Synapsen erhält und entsprechend in der Lage ist, das gesamte Netz zu simulieren und Fire-Events auszugeben. Um die Performance des neuronalen Netzes durch den Simulator zu messen, wird eine Simulationsumgebung eingebunden und ein Lern-Algorithmus implementiert. In dieser Arbeit wird sich auf die Reinforcement Learning Methode Random-Search konzentriert sowie auf die Optimierungsmethode durch Gewichten der entsprechenden Synapsen.

5.1 Performance implementierter Algorithmen

Schnelligkeit der Ausführung von Algorithmen und ganzen Skripten ist in dieser Anwendung von großer Relevanz. Da der Simulator von Grund auf darauf ausgerichtet wurde, später rechenintensive Simulationen von Parametern zu durchlaufen, wurde bereits in der Auswahl der zusätzlich genutzten Pakete darauf geachtet, dass diese performant und vorzugsweise direkt in C implementiert wurden.

Anfangen bei den Berechnungsmodulen in der Datei `lif.py` wird für komplexere mathematische Operationen die Erweiterung `NumPy` aus dem bekannten Python-Paket `SciPy` genutzt. Weiterhin werden Schleifen und If-Abfragen ohne Redundanzen und unnötige Befehle implementiert, um in der höheren Abstraktionsebene einen einwandfreien Aufruf zu garantieren. Nach erfolgreichen Tests der implementierten Funktionen wurde das Framework für den Simulator erstellt. Genutzte Pakete wie `matplotlib` oder `hickle` sind ebenfalls für ihre Schnelligkeit und einfache Handhabung ausgewählt worden. Des Weiteren können hier die bereits implementierten Funktionen zur Berechnung von Synapsenströmen und Membranpotentialen einfach importiert werden.

Letztendlich ist die Ausführung des Suchalgorithmus `RandomSearch` sowie des Optimierers `Weights` ausschlaggebend. Diese Algorithmen wurden im Laufe der Implementierung immer

wieder optimiert und verbessert, sodass eine zuverlässige Simulation mit effizienten Laufzeiten möglich wird. Bei festen Simulationszeiten werden auf der bereits vorgestellten virtuellen Instanz folgende Ergebnisse erzielt (Stichprobenartig aufgelistet). Diese Simulationen wurden

<i>Zeitstempel</i>	<i>Reward</i>	<i>Laufzeit</i>	<i>Anz. Simulationen</i>
20180815_10-40-23	26	2 Std.	39.006
20180816_01-50-01	123	12 Std.	10.509.904
20180816_01-52-01	185	12 Std.	10.536.512
20180818_02-48-01	200	12 Std.	10.852.326

Tabelle 5.1: Parametersuche durch Algorithmus `RandomSearch`.

<i>Zeitstempel</i>	<i>Reward</i>	<i>Laufzeit</i>	<i>Anz. Simulationen</i>
20180815_11-21-46	56	1 Std.	5.927
20180816_13-50-01	149	12 Std.	3.715.008
20180816_13-52-01	200	12 Std.	3.686.723
20180818_...	200	12 Std.	123.

Tabelle 5.2: Optimierung durch Algorithmus `Weights`.

ausnahmslos auf derselben virtuellen Instanz (teilweise parallel) ausgeführt. Die genauen Spezifikationen wurden in Sektion 4.2 bereits genauer beschrieben. Auffällig ist die unterschiedliche Anzahl an Simulationen bei gleichbleibender Zeit zwischen dem Suchalgorithmus `RandomSearch` und dem Optimierungsalgorithmus `Weights`. Im Schnitt werden bei der Parametersuche 10 Mio. Simulationen in einem Zeitraum von 12 Std. erfasst. Die nachgelagerte Optimierung durch den Algorithmus `Weights` ist jedoch rechenintensiver und erfasst innerhalb 12 Std. lediglich ein Drittel: 3,7 Mio. Simulationen.

Letztendlich zeigen diese Daten, dass die implementierten Algorithmen in der Lage sind, dauerhafte Simulationen mit guten Ergebnissen zu erzielen. Durch kleinere Verbesserungen und Veränderungen am Code erzielte der Parametersuchlauf mit dem Zeitstempel `20180818_02-48-01` als erste Mal einen Reward von 200. Dieses Ergebnis beweist die Funktionsweise des Simulators und hält das Pendel in 200 von 200 Simulationsschritten erfolgreich aufrecht. Eine Animation dieser Parameter wird in Appendix ?? genauer erläutert und veranschaulicht.

5.2 Limitationen und Alternativen von Algorithmen

Die bereits vorgestellten Algorithmen `RandomSearch` als Such- und `Weights` als Optimierungsalgorithmus führen zwar mit viel Rechenleistung und hohen Simulationszeiten zu guten und verlässlichen Ergebnissen, sind jedoch im Grunde ineffizient.

5.2.1 Analyse bereits bestehender Algorithmen

RandomSearch generiert Vektoren mit zufälligen Parametern innerhalb einer gegebenen Gleichverteilung und wendet diese auf die Simulationsumgebung an. Der Reward am Ende einer jeden Simulation sagt etwas über die Güte dieser generierten Parameter aus. Ist der Reward hoch, so werden die Parameter gespeichert, fällt der Reward geringer als der bisher beste Reward, wird diese Simulation verworfen. So baut sich ein High-Score-System aus und nach Ablauf der Simulationszeit werden die Parameter mit dem höchsten erreichten Reward gespeichert. Wie darüber

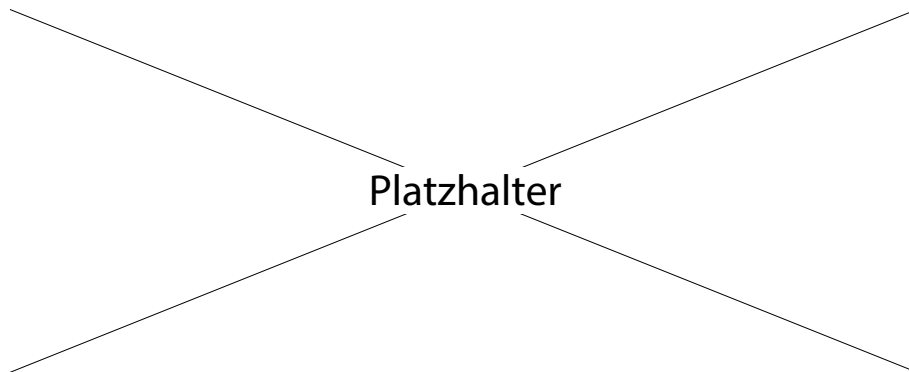


Abb. 5.1: Flowchart des Algorithmus **RandomSearch**.

hinaus in Abb. C.1 noch einmal verdeutlicht, werden Parameter durch den simplen Input des Rewards variiert und gefunden.

Nach Anwendung der Parametersuche durch **RandomSearch** wird eine Optimierung des neuronalen Netzes durch **Weights** durchgeführt. Die gefundenen Parameter sind unter Umständen noch nicht Perfekt gewählt oder verursachen vereinzelt Probleme, welche die Simulation inkonsistent werden lassen. Bei der Optimierung durch Gewichtung der bestehenden Synapsen könnten nachträglich Parameter beeinflusst und gewisse Wege im neuronalen Netz feiner eingestuft werden. In Abb. 5.2 wird der gesamte Programmablauf noch einmal verdeutlicht. In Appendix ??

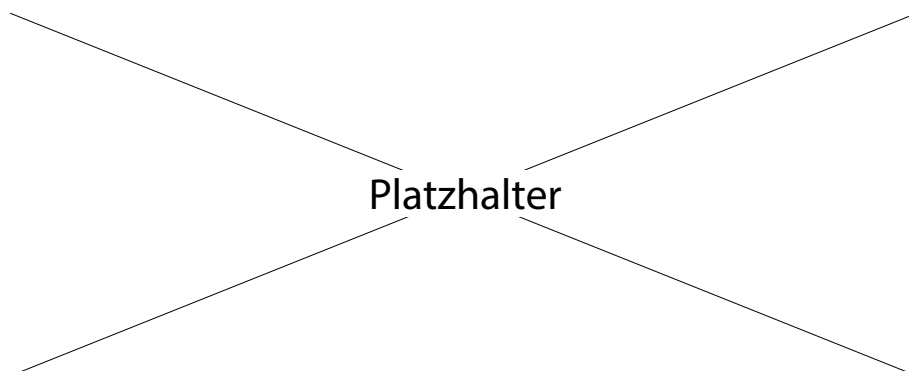


Abb. 5.2: Flowchart des Algorithmus **Weights**.

werden die Ergebnisse der besten Simulationsläufe genauer beschrieben.

Eine weitere Optimierung wurde eingeführt, um das neuronale Netz robuster und effizienter zu gestalten. Anstatt der gesamten 46 Parameter für Nervenzellen, Synapsen und Gap-Junctions

werden jeweils nur die Hälfte der benötigten Parameter simuliert und dupliziert. Dies sorgt für eine symmetrische Verteilung von zufällig generierten Parametern und einer noch effizienteren Simulation. Aufgrund der symmetrischen Architektur des gegebenen neuronalen Netzes ist diese Methode zulässig und führt zu sehr guten Ergebnissen.

5.2.2 Alternative Such- und Optimierungsalgorithmen

Wie bereits in Sektion 3.3 vorgestellt, existieren bereits viele gute Algorithmen zur Parametersuche und -optimierung von künstlich erzeugten oder gegebenen neuronalen Netzen. Doch die Implementierung dieser Algorithmen besonders auf die hohe Anzahl an zu variierenden Parametern stellt eine hohe Anforderung dar. Klassische Optimierungsverfahren über Kostenfunktionen lassen sich zwar aufstellen (wie in ??) kurz gezeigt, können aber durch die Anzahl an Nervenzellen, Synapsen und Gap-Junctions nicht weiter optimiert werden.

Einzig die Optimierung durch Gewichtung von Synapsen und Gap-Junctions kann mit bekannten Algorithmen und den Input des Rewards effizienter als durch RandomSearch umgesetzt werden.

5.3 Vergleich zu bestehenden Systemen

Vergleich zu bestehenden Systemen aus der Regelungstechnik (nötig?).

5.4 Zusammenfassung

5.5 Ausblick


Anhang A

Programmcode Python

A.1 LIF-Modell

Anhang B

Datenblatt Programm: TW-Circuit

<p>Name _____ TW Circuit</p>	<p>QR-Code</p> 
<p>Kurzinfo _____ Lorem Ipsum Dolor sit amen.</p>	<p>Dateibaum _____</p> <p>TW Circuit</p> <ul style="list-style-type: none">docsinformationmodules<ul style="list-style-type: none">inspect.pylif.pyparameters.pyrandom_search_v2.pyvisualize.pyweights.pyparameter_dumpsweight_dumpsmain.py
<p>Dependencies _____</p> <ul style="list-style-type: none">• NumPy aus dem SciPy-Package Numerische Berechnungen und Matrixoperationen.• matplotlib aus dem SciPy-Package Plots und Grafiken anzeigen und exportieren.• OpenAI Gym Simulationsumgebung CartPole_v0• hickle Performantes Speichern im HDF5-Format• os, time, datetime	

Anhang C

Parameter mit guten Simulationsergebnissen

Nachfolgend werden verschiedene Simulationsläufe mit guten Ergebnissen im Detail vorgestellt. Dabei wird besonders auf die Parameter und Gewichte des neuronalen Netzes wert gelegt, da diese durch Simulationen gefunden wurden.

Folgende Konvention wird eingehalten, um die Ergebnisse anschaulich darzustellen:

Parameter der Nervenzellen:

$$\mathbf{C}_m = (C_{AVA} \ C_{AVD} \ C_{PVC} \ C_{AVB}) , \quad (\text{C.1})$$

$$\mathbf{G}_{Leak} = (G_{AVA} \ G_{AVD} \ G_{PVC} \ G_{AVB}) , \quad (\text{C.2})$$

$$\mathbf{U}_{Leak} = (U_{AVA} \ U_{AVD} \ U_{PVC} \ U_{AVB}) . \quad (\text{C.3})$$

Parameter der Synapsen und Gap-Junctions:

$$\boldsymbol{\sigma} = (\sigma_1 \dots \sigma_{16}) , \quad (\text{C.4})$$

$$\mathbf{w} = (w_1 \dots w_{16}) , \quad (\text{C.5})$$

$$\hat{\mathbf{w}} = (\hat{w}_1 \ \hat{w}_2) . \quad (\text{C.6})$$

Gewichte der Synapsen und Gap-Junctions:

$$\mathbf{g} = (g_1 \dots g_{18}) . \quad (\text{C.7})$$

Weitere Parameter und Einheiten sind der Tabelle ?? zu entnehmen.

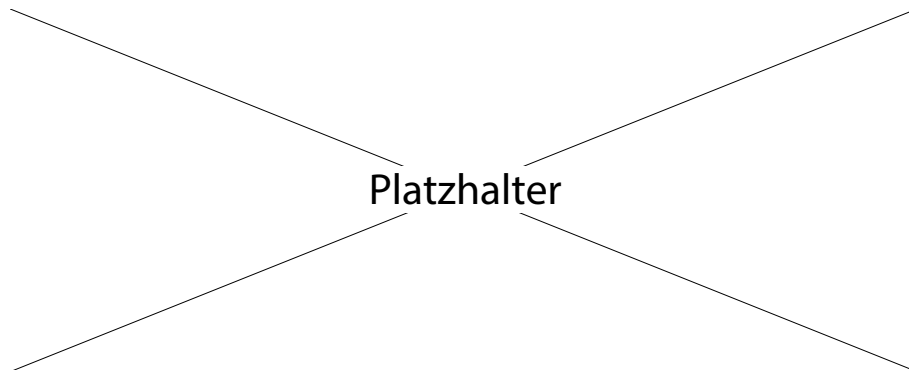


Abb. C.1: Neuronales Netz mit Legende für Gewichte

<i>Zeitstempel</i>	<i>Algorithmus</i>	<i>Reward</i>	<i>Simulation (Dauer, Anz.)</i>	<i>Parameter</i>
20180817_01-56-01	RandomSearch - Parameter	131	12h, ca. 1 Mio. Simulationen	Parameter der Nervenzellen $\mathbf{C_m} = (C_{AVA}, C_{AVD}, C_{PVC}, C_{AVB}),$ $\mathbf{G_{Leak}} = (G_{AVA}, G_{AVD}, G_{PVC}, G_{AVB}),$ $\mathbf{U_{Leak}} = (U_{AVA}, U_{AVD}, U_{PVC}, U_{AVB}).$ Parameter der Synapsen & GJ $\boldsymbol{\sigma} = (\sigma_1, \dots, \sigma_{16}),$ $\mathbf{w} = (w_1, \dots, w_{16}),$ $\hat{\mathbf{w}} = (\hat{w}_1, \hat{w}_2).$
20180817_13-56-01	Weights - Gewichte	200	12h, ca. 400.000 Simulationen	Gewichte der Synapsen $\mathbf{g} = (g_1, \dots, g_{18}).$

Tabelle C.1: Table caption

Literaturverzeichnis

1. Chalfie M, Sulston JE, White JG, Southgate E, Thomson JN, Brenner S (1985) The neural circuit for touch sensitivity in *Caenorhabditis elegans*. *The Journal of Neuroscience* 5(4):956–964
2. Hasani RM, Beneder V, Fuchs M, Lung D, Grosu R (????) Sim-ce: An advanced simulink platform for studying the brain of *Caenorhabditis elegans*
3. Lechner M, Grosu R, Hasani RM (2017) Worm-level control through search-based reinforcement learning
4. Rojas R (1996) *Theorie der neuronalen Netze: Eine systematische Einführung* (Springer-Lehrbuch) (German Edition). Springer, URL <https://www.amazon.com/Theorie-neuronalen-Netze-systematische-Springer-Lehrbuch/dp/3540563539?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=3540563539>
5. Strogatz SH (1994) *Nonlinear Dynamics And Chaos: With Applications To Physics, Biology, Chemistry And Engineering* (Studies in Nonlinearity). Studies in nonlinearity, Perseus Books, URL <https://books.google.de/books?id=PHmED2xxrE8C>
6. Wicks SR, Roehrig CJ, Rankin CH (1996) A dynamic network simulation of the nematode tap withdrawal circuit: Predictions concerning synaptic function using behavioral criteria. *J Neurosci* 16(12):4017, URL <http://www.jneurosci.org/content/16/12/4017.abstract>
7. Wilinski J (2018) Bachelorarbeit repository. Online, URL <https://github.com/J0nasW/BA>
8. Wulfram Gerstner RNLP Werner M Kistler (2014) *Neuronal Dynamics From Single Neurons to Networks and Models of Cognition*, 1st edn. Cambridge University Press, Cambridge CB2 8BS, United Kingdom

Erklärung

Ich versichere, dass ich die Bachelor-Arbeit

Eine Anwendung des Reinforcement Learning zur Regelung dynamischer Systeme

selbständig und ohne unzulässige fremde Hilfe angefertigt habe und dass ich alle von anderen Autoren wörtlich übernommenen Stellen, wie auch die sich an die Gedankengänge anderer Autoren eng anlehnenden Ausführungen, meiner Arbeit besonders gekennzeichnet und die entsprechenden Quellen angegeben habe.

Mir ist bekannt, dass die unter Anleitung entstandene Bachelor-Arbeit, vorbehaltlich anders lautender Vereinbarungen, eine Gruppenleistung darstellt und in die Gesamtforschung der betreuenden Institution eingebunden ist. Daher darf keiner der Miturheber (z.B. Texturheber, gestaltender Projektmitarbeiter, mitwirkender Betreuer) ohne (schriftliches) Einverständnis aller Beteiligten, aufgrund ihrer Urheberrechte, auch Passagen der Arbeit weder kommerziell nutzen noch Dritten zugänglich machen. Insbesondere ist das Arbeitnehmererfindergesetz zu berücksichtigen, in dem eine Vorveröffentlichung patentrelevanter Inhalte verboten wird.

Kiel, 08. September 2018

Jonas Helmut Wilinski