Jonas Helmut Wilinski

Eine Anwendung des Reinforcement Learning zur Regelung dynamischer Systeme

Bachelor-Arbeit

Stand: September 2018

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

Inhaltsverzeichnis

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* [7]. 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-Architektur[3] 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 Axom.*

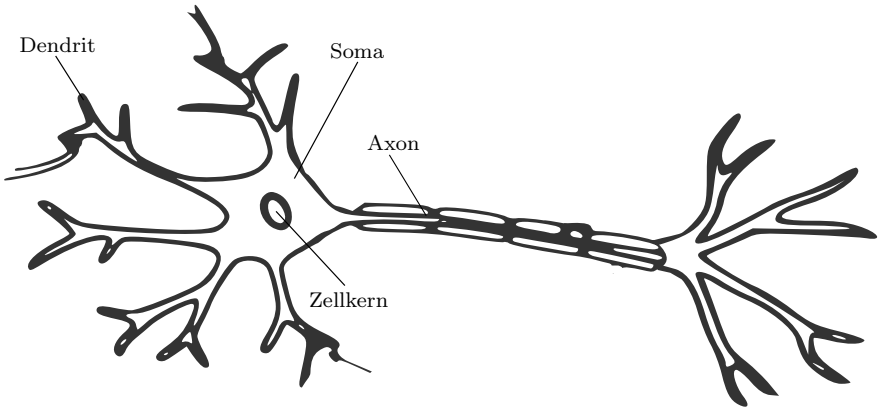


Abb. 1.1: Schematische Darstellung einer Nervenzelle bestehend aus Dendrit, Soma und Axon.

- 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 Ionentransport 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.

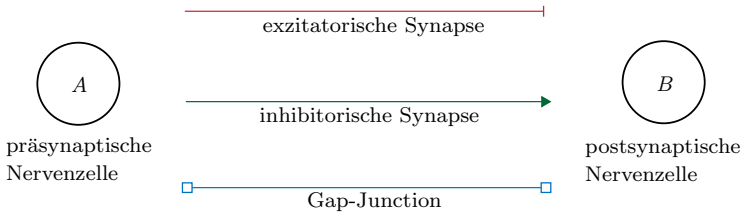


Abb. 1.2: Darstellung von verschiedenen Synapsen-Typen.

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

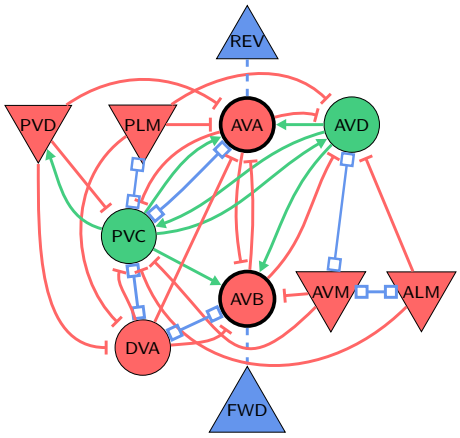


Abb. 1.3: TW-Neuronal-Circuit nach Lechner et al. [3]

Stimulus - sprich einer Berührung - ausgesetzt, so schnellt 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:

Umgebungsvariable	*Typ*	*Posivite Sensor-Neurone*	*Negative Sensor-Neurone*
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 verbunden. In jeder internen Nervenzelle findet ein Integrationsprozess der jeweiligen anliegenden Ströme aus Stimulus (I_{Stim}), anderen chemikalischen Synapsen (I_{Syn}) und Gap-Junctions (I_{Gap}) statt. Durch das Leaky Integrate and Fire - Modell kann das Membranpotential zu jedem beliebigen Zeitpunkt bestimmt und ein mögliches Feuer-Event vorhergesagt werden. Eine Nervenzelle feuert ein Signal, wenn das Membranpotential einen Treshold $\theta = -20mV$ 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.

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 neuroalen 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 'Treshold' θ beschrieben wird. Bei Überschreitung 'feuert' die Nervenzelle und die Informationen gelangen über Synapsen zu nahegelegenen Neuronen.
Technisch lässt sich dieses Verhalten als ein Schaltbild wie in Abb. 2.1 darstellen. Die Zellmembran wird durch den Kondensator repräsentiert, welcher durch eingehende Stimuli-, Synapsen- und Gap-Junction-Ströme geladen wird und bei einem gewissen Treshold augenblicklich entlädt. Da die Zellmembran jedoch als Isolator nicht perfekt ist, wird ein Widerstand in Reihe mit der Ruhespannung u_{rest} geschaltet (siehe [7] Kap. 1.3.1). Um nun den Spannungsverlauf der

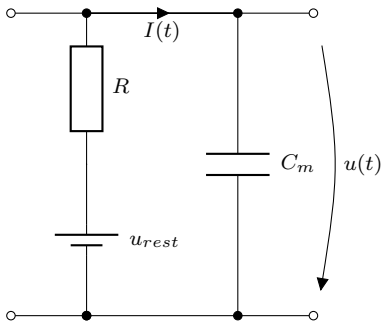


Abb. 2.1: Ersatzschaltbild der Zellmembran

Zellmembran innerhalb der Nervenzelle zu berechnen, ist durch das Modell folgende lineare Differentialgleichung erster Ordnung gegeben:

$$\frac{du_i(t)}{dt} = \frac{G_{Leak}(U_{Leak} - u_i(t)) + \sum_{i=1}^{n} I_{in}^{(1)}}{C_m} \tag{2.1}$$

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

$$I_{in} = I_{Stimuli} + I_{Syn} + I_{Gap} \tag{2.2}$$

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

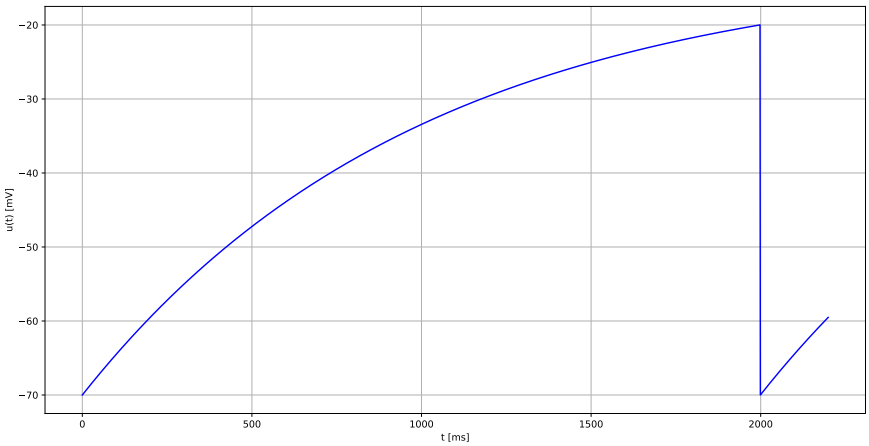


Abb. 2.2: Graphische Darstellung des Membranpotentials durch das Leaky Integrate and Fire - Modell

Die anliegenden Synapsenströme sind durch folgenden formularen Zusammenhang zu berechnen:

$$I_{Syn} = \frac{w}{1 + \mathrm{e}^{\sigma(u_{pre}(t)+\mu)}}(E - u_{post}(t)) \tag{2.3}$$

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

$$I_{Gap} = \hat{w}(u_{post}(t) - u_{pre}(t)) \tag{2.4}$$

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 formularen 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:

$$S_{positiv} := \begin{cases} -70mV & x \leq 0 \\ -70mV + \frac{50mV}{x_{min}} x & 0 < x \leq x_{min} \\ -20mV & x > x_{max} \end{cases} \tag{2.5}$$

$$S_{negativ} := \begin{cases} -70mV & x \geq 0 \\ -70mV + \frac{50mV}{x_{min}} x & 0 > x \geq x_{min} \\ -20mV & x < x_{max} \end{cases} \tag{2.6}$$

$x \in [x_{min}, x_{max}]$ 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. Genaueres 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 2.1, 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:

$$x_{n+1} = x_n + f(x_n)\Delta t \tag{2.7}$$

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:

$$\tilde{x}_{n+1} = x_n + f(x_n)\Delta t \tag{2.8}$$
$$x_{n+1} = x_n + \tfrac{1}{2}[f(x_n) + f(\tilde{x}_{n+1})]\Delta t \tag{2.9}$$

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:

$$\begin{aligned} k_1 &= f(x_n)\Delta t \\ k_2 &= f(x_n + \tfrac{1}{2}k_1)\Delta t \\ k_3 &= f(x_n + \tfrac{1}{2}k_2)\Delta t \\ k_4 &= f(x_n + k_3)\Delta t \end{aligned} \tag{2.10}$$

$$x_{n+1} = x_n + \tfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{2.11}$$

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 2.1 resultiert in folgende Berechnung, welche direkt implementiert werden kann:

$$\begin{aligned} k_1 &= \frac{(G_{leak}(U_{leak} - u_i(t)) + (I_{Stimuli} + I_{Syn} + I_{Gap}))}{C_m}\Delta t \\ k_2 &= \frac{(G_{leak}(U_{leak} - (u_i(t) + \tfrac{1}{2}k_1)) + (I_{Stimuli} + I_{Syn} + I_{Gap}))}{C_m}\Delta t \\ k_3 &= \frac{(G_{leak}(U_{leak} - (u_i(t) + \tfrac{1}{2}k_2)) + (I_{Stimuli} + I_{Syn} + I_{Gap}))}{C_m}\Delta t \\ k_4 &= \frac{(G_{leak}(U_{leak} - (u_i(t) + k_3)) + (I_{Stimuli} + I_{Syn} + I_{Gap}))}{C_m}\Delta t \end{aligned} \tag{2.12}$$

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

$$u_{i+1}(t) = u_i(t) + \tfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{2.13}$$

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. Das gesamte Modul zur Berechnung von Strömen und Spannungen pro Zeiteinheit wird wie folgt implementiert:

Algorithmus 1: compute

Input : $\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C_m}, \boldsymbol{G_{Leak}}, \boldsymbol{U_{Leak}}, \boldsymbol{\sigma}, \boldsymbol{w}, \hat{\boldsymbol{w}}$
Output: Simulationsinformation (information.txt), Parameter-Dump als .hkl Datei

```
for i ← 1 to 4 do
    for j ← 1 to 4 do
        I_interi,j = I_syn_calc(x[i], x[j], E, w[k], σ[k], μ)
        I_sensori,j = I_syn_calc(u[i], u[j], E, w[k], σ[k], μ)
        k ← k + 1 I_gapi,j = I_gap_calc(x[i], x[j], ŵ[k])
    end
end
return information.txt, parameter_dump.hkl, date, best_reward
```

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. Der Lernprozess erfolgt, indem der Agent eine Aktion tätigt, welche er dann durch die resultierenden Ergebnisse einschätzt und basierend auf dieser immer größer werdenden Datenbasis neue Aktionen tätigt.

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 Behlohnungssysteme durch die Umwelt zu trainieren. Entsprechend benötigen diese Algorithmen mehr Rechenleistung (je nach Aufgabenstellung). Sie versuchen, in einer Anhäufung von Daten Strukturen zu erkennen, welche von stochastischem Rauschen abweichen. Neuronale Netze ori-

entieren 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).

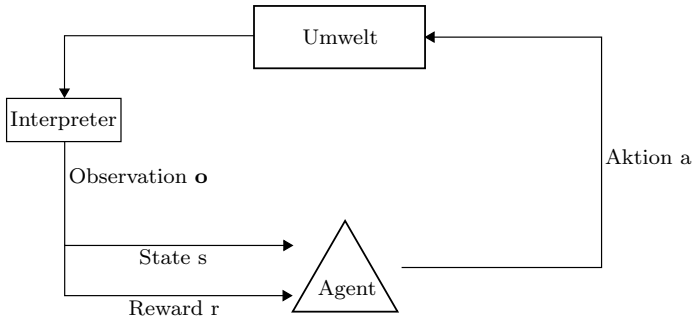


Abb. 3.1: Graphische Darstellung des Reinforcement Learning Algorithmus

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 **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 Episode höher und der Agent ist in der Lage gewisse Parameter der Simulation entsprechend des jeweiligen Observationsparameters anzupassen.

3.2 Anwendung auf Modelle neuronaler Netze

Die klassische Anwendung von Deep Learning Algorithmen auf künstlich erstellte neuronale Netze mit vielen s.g. '"Hidden Layers"'(Ebenen zwischen Ein- und Ausgang mit hoher Anzahl an Neuronen) findet im Rahmen dieser Bachelorarbeit nicht statt. Jedoch können viele Methoden und Algorithmen abgewandelt werden, um ein bereits bestehendes, neuronales Netz aus der Natur zu nutzen und auf Probleme der Regelungstechnik anzuwenden. Folglich befassen wir uns mit einem neuronalen Netz, welches ein Layer (zu Deutsch: 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 Observa-

tionsvektor **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 Random Search 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

3.3.2 Gradient Policies

3.3.3 Genetische Algorithmen

3.3.4 Random Search

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

Parameter-Typ	*Parameter*	*Beschreibung*	*Grenzen*
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 Potassiummengen 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 excitatorische Synapsen und x Gap-Junctions.

3.4 Implementierung

Algorithmus 2: Das LIF-Modell

Input : $u, u_{rest}, t, \vartheta, R, C, I_0$
Output: Array $u(t)$ mit $t = 1, 2, 3, ...$

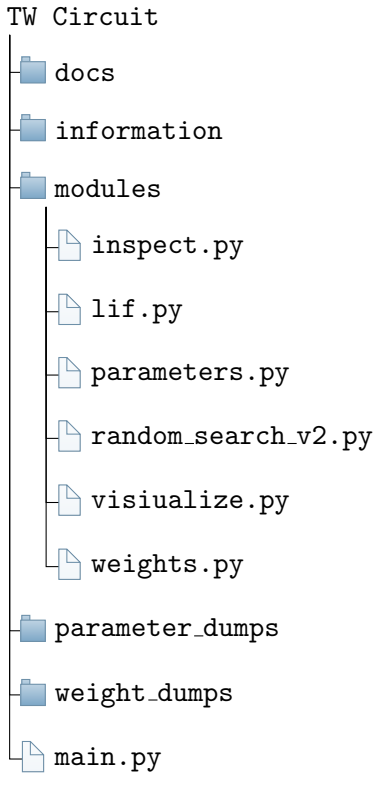
```
for i ← 0 to t_max do
    i als Zähl-Variable
    if u ≤ ϑ then
        /* Aufaddieren, bis der Threshold ϑ erreicht ist */
        Berechne momentane Spannung u bei t = i
        Erweitere das Array u_array um aktuelle Spannung u i hochzählen i+ = 1
    else
        /* Treshold ϑ ist erreicht, setze u auf 0 zurück */
        i = 0 Berechne momentane Spannung u bei t = 0
        Erweitere das Array u_array um aktuelle Spannung u i hochzählen i+ = 1
    end
end
return hhh
```

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. Die Module sowie diverse Dokumentationen und Informationen sind auf meiner GitHub-Repository[1] [6] zu finden.

4.1 Aufbau des Programms

Das Paket TW Circuit beinhaltet folgende Module:



- 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.

[1] https://github.com/J0nasW/BA

4.2 Implementierung der Suchalgorithmen

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 erzeugen, wird das bekannte Paket `numpy` verwendet. Nach Aufruf des Algorithmus `random_parameters` wird eine Simulation mit

Algorithmus 3: 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:`
C_m = np.random.uniform(low = 0.01, high = 1, size = (1,Anz. Nervenzellen))
G_{leak} = np.random.uniform(low = 0.05, high = 5, size = (1,Anz. Nervenzellen))
U_{leak} = np.random.uniform(low = -70, high = 0, size = (1,Anz. Nervenzellen))
`// Für Synapsen:`
σ = np.random.uniform(low = 0.05, high = 0.5, size = (1,Anz. Synapsen))
w = np.random.uniform(low = 0, high = 3, size = (1,Anz. Synapsen))
$\hat{w}$ = np.random.uniform(low = 0, high = 03, size = (1,Anz. Gap-Junctions)) **return** $C_m, G_{Leak}, U_{Leak}, \sigma, w, \hat{w}$

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

Algorithmus 4: random_search_v2

Input : Simulationszeit
Output: Simulationsinformation (information.txt), Parameter-Dump als .hkl Datei
action = episodes = best_reward = 0
env = gym.make('CartPole-v0')
while *True* **do**
  initialize(Default_U_leak)
  episodes ← episodes + 1
  $C_m, G_{Leak}, U_{Leak}, \sigma, w, \hat{w}$ = random_parameters()
  reward = run_episode($C_m, G_{Leak}, U_{Leak}, \sigma, w, \hat{w}$) ; `// Simulation mit neuen Parametern - Siehe Sec. 4.3`
  if *reward* $\geq$ *best_reward* **then**
    Set best_reward ← reward Result = $[C_m, G_{Leak}, U_{Leak}, \sigma, w, \hat{w}]$; `// Für Parameter-Dump`
    if *reward* $\geq$ *200* **then**
      break; `// Exit-Argument, wenn Reward von 200 erreicht wurde`
    end
  end
  if *elapsed_time* > *simulation_time* **then**
    break ; `// Exit-Argument, um genaue Laufzeiten zu erzielen`
  end
end
return *information.txt, parameter_dump.hkl, date, best_reward*

der Synapsenströme sowie Membranpotentiale und die Simulation findet in der Methode `run_episode` statt und wird in Sektion 4.3 präziser erläutert.

4.2.2 Suchalgorithmus Weights

4.2.3 Simulation in der Google Cloud Platform®

4.3 Simulationsumgebung: OpenAI Gym

main.py OpenAI Gym

4.4 Visualisierung und Auswertung

visiualize.py

Algorithmus 5: Das LIF-Modell

Input : $u, u_{rest}, t, \vartheta, R, C, I_0$
Output: Array $u(t)$ mit $t = 1, 2, 3, ...$

for $i \leftarrow 0$ **to** t_{max} **do**
  i als Zähl-Variable
  if $u \leq \vartheta$ **then**
    `/* Aufaddieren, bis der Threshold` ϑ `erreicht ist */`
    Berechne momentane Spannung u bei $t = i$
    Erweitere das Array u_{array} um aktuelle Spannung u i hochzählen $i+ = 1$
  else
    `/* Treshold` ϑ `ist erreicht, setze` u `auf` 0 `zurück */`
    $i = 0$ Berechne momentane Spannung u bei $t = 0$
    Erweitere das Array u_{array} um aktuelle Spannung u i hochzählen $i+ = 1$
  end
end
return u_{array}

4.5 Sonstige Implementierung

inspect.py parameters.py parameter und weight dumps + hickle

4.6 Steuerung und Zusammenfassung

Kapitel 5
Performance & Auswertung

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.
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.

5.1 Performance implementierter Algorithmen

5.2 Limitationen und Alternativen von Algorithmen

5.3 Vergleich zu bestehenden Systemen

5.4 Zusammenfassung

Algorithmus 6: Das LIF-Modell

Input : $u, u_{rest}, t, \vartheta, R, C, I_0$
Output: Array $u(t)$ mit $t = 1, 2, 3, ...$
for $i \leftarrow 0$ **to** t_{max} **do**
  i als Zähl-Variable
  if $u \leq \vartheta$ **then**
    `/* Aufaddieren, bis der Threshold` ϑ `erreicht ist */`
    Berechne momentane Spannung u bei $t = i$
    Erweitere das Array u_{array} um aktuelle Spannung u i hochzählen $i+ = 1$
  else
    `/* Treshold` ϑ `ist erreicht, setze` u `auf 0 zurück */`
    $i = 0$ Berechne momentane Spannung u bei $t = 0$
    Erweitere das Array u_{array} um aktuelle Spannung u i hochzählen $i+ = 1$
  end
end
return u_{array}

Anhang A
Programmcode Python

A.1 LIF-Modell

```
tau_m = R * C
u_array = np.array([])

for t in range(0, 100):
        if u <= v:
                u = u_rest + (R * I_0 * (1 - np.exp(-(i / tau_m))))
                u_array = np.append(u_array, u)
                i += 1
        else:
                i = 0
                u = u_rest + (R * I_0 * (1 - np.exp(-(i / tau_m))))
                u_array = np.append(u_array, u)
                i += 1
return u_array
```

Anhang B

Aufbau Modul

- main.py
- modules

  visiualize.py

  ...

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. Wilinski J (2018) Bachelorarbeit repository. Online, URL https://github.com/J0nasW/BA

7. 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