

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

<b>Abstract und Kurzfassung</b> .....	1
<b>1 Einleitung</b> .....	3
<b>2 Grundlagen der neuronalen Netze</b> .....	5
2.1 Grundlegende Berechenbarkeitsmodelle .....	5
2.2 Die biologische Nervenzelle .....	6
2.3 Das biologische neuronale Netz .....	8
2.4 Das symmetrische neuronale Netz .....	10
<b>3 Leaky Integrate and Fire und simulative Modelle neuronaler Netze</b> .....	11
3.1 Das Leaky Integrate and Fire - Modell .....	11
3.2 Anwendung auf Modelle neuronaler Netze .....	13
3.3 Zuverlässigkeit und Limitationen .....	14
3.4 Implementierung .....	14
<b>4 Reinforcement Learning - Lernen mit Belohnung</b> .....	19
4.1 Reinforcement Learning - eine Abwandlung des Deep Learning .....	19
4.2 Anwendung auf Modelle neuronaler Netze .....	21
4.3 Verschiedene Suchalgorithmen .....	21
4.3.1 Q-Learning .....	22
4.3.2 Gradient Policies .....	23
4.3.3 Genetische Algorithmen .....	23
4.3.4 Random Search .....	24
<b>5 Implementierung des TW-Netzes</b> .....	27

5.1	Aufbau des Programms .....	27
5.2	Implementierung der Suchalgorithmen .....	28
5.2.1	Suchalgorithmus RandomSearch .....	29
5.2.2	Optimierungsalgorithmus Weights .....	30
5.2.3	Simulation in der Google Cloud Platform® .....	30
5.3	Simulationsumgebung: OpenAI Gym .....	32
5.4	Visualisierung und Auswertung .....	34
5.5	Sonstige Implementierung .....	36
5.5.1	Zentraler Ort für Parameter .....	36
5.5.2	Speicherung von Daten .....	36
5.5.3	Dateiinspektion .....	36
<b>6</b>	<b>Performance &amp; Auswertung .....</b>	<b>37</b>
6.1	Performance implementierter Algorithmen .....	37
6.2	Limitationen und Alternativen von Algorithmen .....	38
6.2.1	Analyse bereits bestehender Algorithmen .....	39
6.2.2	Alternative Such- und Optimierungsalgorithmen .....	39
6.3	Zusammenfassung .....	40
6.4	Ausblick .....	41
<b>A</b>	<b>UML-Diagramme wichtiger Algorithmen .....</b>	<b>43</b>
A.1	RandomSearch .....	43
A.2	Weights .....	44
<b>B</b>	<b>Datenblatt Programm: TW Circuit .....</b>	<b>45</b>
<b>C</b>	<b>Parameter mit guten Simulationsergebnissen .....</b>	<b>47</b>
	<b>Literaturverzeichnis .....</b>	<b>51</b>

# Abstract und Kurzfassung

## Abstract

This bachelor thesis implements a way to achieve a reliable and stable control of a dynamic system through approaches of reinforcement learning. For a neuronal network, the „Touch Withdrawal Circuit“ of the worm *C. Elegans* is examined in great detail and the structures are transformed into a simulator. As a simulation environment, the inverted pendulum is being used with one degree of freedom (1 DOF).

To simulate the neural network and guarantee reliable control for the inverted pendulum, a simulator is being developed and implemented using the programming language `Python`. Using the well known Leaky Integrate and Fire model, simulation of internal neural dynamics and processing information within the network is made possible. Furthermore, Parameters of the network are found using reinforcement learning algorithms and applied to the environment `CartPole_v0` from OpenAI Gym.

The result of this work shows, that it is possible to implement a functional simulator for biological neural networks and to link it with methods of reinforcement learning. After computing multiple simulations, suitable Parameters for the network, which ensure stable control of the inverse pendulum, are found. An application to other simulation environments or with similar neural networks is also possible due to the structure of the simulator.

## Kurzfassung

Ziel dieser Bachelorarbeit ist es, durch Ansätze des Reinforcement Learning, sowie unter Nutzung biologischer neuronaler Netze, eine zuverlässige und stabile Regelung eines dynamischen Systems zu erzielen. Als neuronales Netz wird der s.g. „Touch Withdrawal Circuit“ des Wurms *C. Elegans* detailliert untersucht und die Strukturen in einen Simulator überführt. Als Simulationsumgebung wird das inverse Pendel mit einem Freiheitsgrad gewählt.

Um das o.g. neuronale Netz zu simulieren und die Regelung auf das inverse Pendel zu übertragen, wird ein Simulator in der Programmiersprache `Python` entwickelt und implementiert. Dieser nutzt das bekannte Leaky Integrate and Fire Modell, um die neuronale Dynamik zu simulieren und Prozesse innerhalb des Netzwerkes darzustellen. Parameter des Netzwerkes werden durch Algorithmen des Reinforcement Learning gefunden und auf die Umgebung `CartPole_v0` (OpenAI Gym) angewendet.

Das Ergebnis dieser Arbeit zeigt, dass es möglich ist, einen funktionsfähigen Simulator für biologische neuronale Netze zu implementieren und diesen mit Methoden des Reinforcement Learning zu koppeln. Durch rechenintensive Simulationen wurden geeignete Parameter für das Netzwerk gefunden, welche eine stabile Regelung des inversen Pendels gewährleisten. Eine Anwendung auf weitere Simulationsumgebungen oder mit ähnlichen neuronalen Netzen ist durch den universalen Aufbau des Simulators ebenfalls möglich.

# Kapitel 1

## Einleitung

Diese Arbeit beschäftigt sich mit Bereichen des Reinforcement Learning und der Anwendung biologischer neuronaler Netze auf Probleme der Regelungstechnik.

Hierzu wird als Basis ein Paper von Lechner et al. „Worm-level Control through Search-based Reinforcement Learning“ [7] herangezogen. Anders als herkömmliche neuronale Netze, welche meist künstlich erzeugt und angewendet werden, findet in dieser Arbeit ein biologisches Netz des Wurms *C. Elegans* Anwendung. Dieses Netz wurde schon durch verschiedene Paper [1] [4] [15] untersucht und vorgestellt. Um dieses Netz zu simulieren, wird auf das Leaky Integrate and Fire Modell verwiesen, welches eine gute Berechnungsgrundlage für Prozesse innerhalb biologischer neuronaler Netze bietet. Zur Berechnung der neuronalen Dynamik müssen charakteristische Größen innerhalb des Netzes berechnet werden:

- Membranpotential  $U_i$  einer Nervenzelle und
- Anliegende Ströme  $I_i$  aus Sensorneuronen, Synapsen und Gap-Junctions.

Um die Informationsverarbeitung innerhalb des Netzes zu nutzen, werden s.g. Sensor- und Motor-Neuronen definiert. Sensor-Neuronen nehmen Größen aus der gegebenen Umwelt auf und übersetzen diese in ein verständliches Format. In diesem Fall werden Größen auf einen Aktionsbereich von  $A \in [-70mV, -20mV]$  übersetzt. Gleichzeitig können Motor-Neuronen den genannten Aktionsbereich  $A$  bspw. auf translatorische Größen anwenden.

Aufgrund des komplexen Berechenbarkeitsmodells biologischer neuronaler Netze müssen für alle eingesetzten Nervenzellen, Synapsen und Gap-Junctions Parameter gefunden werden, welche eine gute und stabile Simulation der gegebenen Umwelt gewährleisten. Hier wird auf die Methode des Reinforcement Learning verwiesen. Durch zufälliges Sampling der geforderten Größen in einem vorher festgelegten Bereich wird eine Vielzahl an Simulationen angesetzt. Ein Belohnungssystem gibt Aufschluss über die Güte der eingesetzten Parameter. Besonders gute Parametersätze werden nach Ablauf der Simulationszeit gespeichert. Durch eine nachgelagerte Optimierung des Netzwerkes mit Gewichtung der Synapsen und Gap-Junctions werden stabile und reproduzierbare Simulationsergebnisse erzielt. Als Simulationsumgebung wird das inverse Pendel genutzt. Dieses wird aus dem bereits bestehenden Framework OpenAI Gym importiert und integriert (`CartPole_v0`). Das inverse Pendel kann im Lernprozess stabilisiert werden, entsprechende Parameter sowie Animationen sind in Anhang C zu finden.





## Kapitel 2

# Grundlagen der neuronalen Netze

Im Laufe der Zeit wurde das Konstrukt der neuronalen Netze erforscht und stark abstrahiert. Dies ermöglicht den Einsatz neuartiger Maschinen und Algorithmen zum Lösen verschiedener Problemstellungen. Um die Performance der klassischen Automaten bzw. Rechenmaschinen zu testen, werden einige Berechenbarkeitsmodelle vorgestellt. Die einzigartige Umsetzung in Netzwerken neuronaler Nervenzellen ermöglicht es, hochkomplexe Aufgaben selbst bei niedrigen Taktfrequenzen durch hohe Parallelität zu bewältigen. Im weiteren Verlauf wird 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* [10] und *Gerstner et al. - Neuronal Dynamics* [16]. Weitere Fachartikel werden im Laufe des Kapitels genannt.

### 2.1 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 entwickelt 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*<sup>1</sup> wurden die berechenbaren Funktionen wie folgt abgegrenzt: „Die berechenbaren Funktionen sind die allgemein rekursiven Funktionen.“

---

<sup>1</sup> Alonzo Church & Alan Turing, 1936

- Das logisch-operationelle Modell (Turing Maschine)

Durch die Turing Maschine<sup>2</sup> war es möglich, neben der mathematischen Herangehensweise an Berechenbarkeitsprobleme, eine mechanische Methode einzusetzen. 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 dem Jahr 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 in der Lage war, alle rekursiven Funktionen zu lösen. Er verfügte über die damals etablierte Von-Neumann-Architektur<sup>3</sup> und wurde von Frederic Calland Williams erbaut.

- Das Modell der Zellautomaten

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

- Das biologische Modell (neuronale Netze)

Neuronale Netze heben sich jedoch 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 auch im ganzen Netzwerk möglich. Jedoch werden im neuronalen Netz keine Programme (wie in Rechenmaschinen üblich) gespeichert, sondern durch die s.g. Netzparameter erlernt. Dieser Ansatz wurde früher durch mangelnde Rechenleistung der konventionellen Computer nicht weiter verfolgt. Heute erfahren wir immer mehr den Aufwind neuester Lernalgorithmen und Frameworks, die das Arbeiten im Bereich Deep Learning, Artificial Intelligence und adaptives Handeln unterstützen und beschleunigen. Weitergehend ist man 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.

## 2.2 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,

---

<sup>2</sup> Alan Turing, 1936

<sup>3</sup> John von Neumann, 1945

einem Zellskelett und einem Zellkern, welcher die chromosomale DNA und somit die Mehrzahl an Genen enthält. Sie treten im menschlichen Körper in verschiedenen Größen und mit unterschiedlichen Fähigkeiten auf. Neuronale Nervenzellen wurden über die Evolution dahingehend ausgeprägt, dass sie Informationen empfangen, verarbeiten und entsenden können. Wie in Abb. 2.1 zu sehen, besteht eine Nervenzelle aus drei Bestandteilen: *Dendrit*, *Soma* und *Axon*.

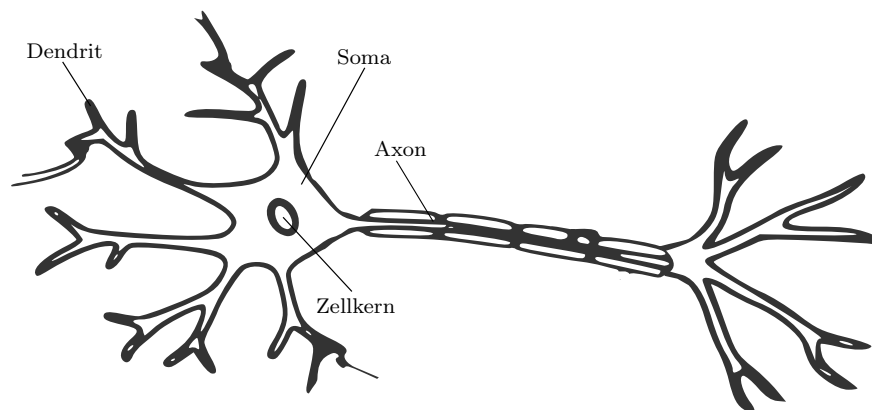


Abb. 2.1: Schematische Darstellung einer Nervenzelle bestehend aus Dendrit, Soma und Axon  
(<https://pixabay.com/de/gehirn-neuron-nerven-zelle-2022398> mit eigenen Erweiterungen - Zugriff 06.08.2018, 16:00h).

- 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 ziehen eine Änderung des Membranpotentials mit sich. 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 weiteren Verlauf dieser Arbeit werden Synapsen nach Abb. 2.2 dargestellt.

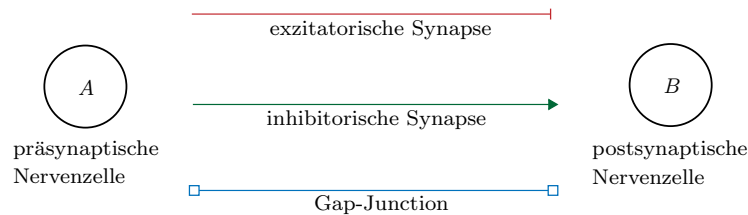


Abb. 2.2: Darstellung von verschiedenen Synapsen-Typen.

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.

## 2.3 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. [7] grafisch dargestellte neuronale Netz des *C. Elegans*, welches den Berührungs-Reflex des Wurms modelliert.

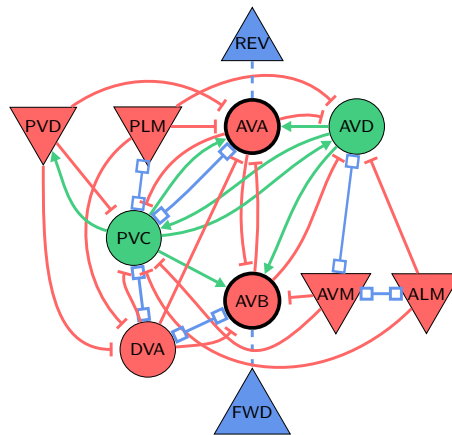


Abb. 2.3: TW-Neuronal-Circuit nach Lechner et al. [7]

Wird der Wurm einem äußeren 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 Rezeptoren 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  $-70mV$  (Ruhepotential - 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	Positive Sensor-Neurone	Negative Sensor-Neurone
$\varphi$	Observation	PLM	AVM
$\dot{\varphi}$	Observation	ALM	PVD
$a$	Action	FWD	REV

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 chemischen 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 Schwellwert („Threshold“)  $\vartheta = -20mV$  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, einen 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 somit ein in sich geschlossenes System mit vier Eingängen und zwei

Ausgängen (man achte auf das Mapping mit positiven und negativen Werten) und formt ein lernfähiges neuronales Netz.

## 2.4 Das symmetrische neuronale Netz

Wie in [15] bereits thematisiert, wird in Abb. 2.3 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. 2.3 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 nicht Berücksichtigt, 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.

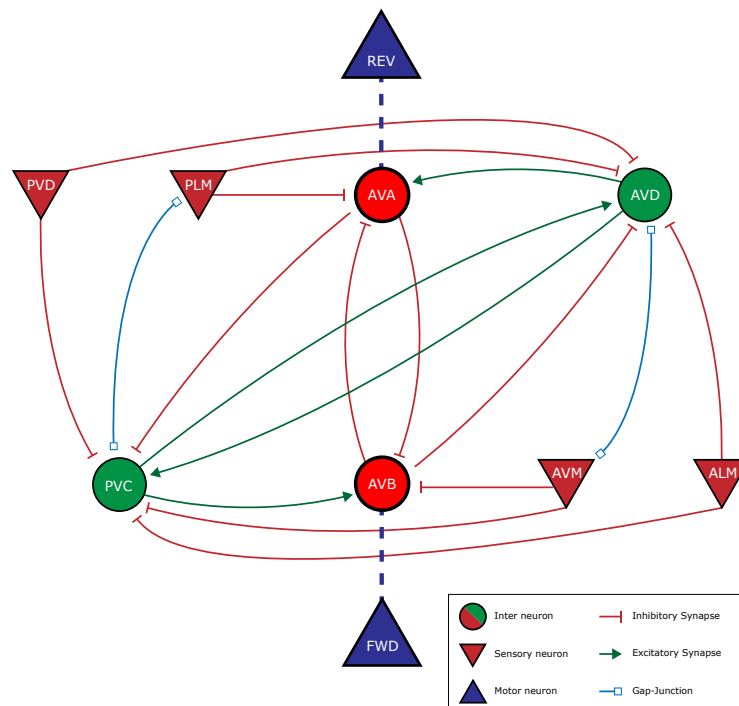


Abb. 2.4: Symmetrisches neuronales Netz des TW-Circuits

Das in Abb. 2.4 dargestellte, symmetrische neuronale Netz des Touch Withdrawal Circuit wird für alle weiteren Analysen und Simulationsläufe verwendet. Die genaue Umsetzung wird in Kapitel 5 weiter erläutert.

## Kapitel 3

# 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 Differenzialgleichung 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.

### 3.1 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“  $\vartheta$  beschrieben wird. Bei Überschreitung dieses Grenzwertes „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 analog eines Kondensators 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. 3.1 darstellen (siehe [16] Kap. 1.3.1). Um nun eine geeignete Differenzialgleichung herzuleiten, wird zuerst das erste Kirchhoffsche Gesetz angewendet

$$I(t) = I_R + I_C. \quad (3.1)$$

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

$$I_R = \frac{U_R}{R} = \frac{U - U_{Leak}}{R}. \quad (3.2)$$

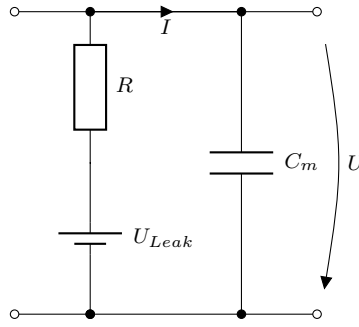


Abb. 3.1: Ersatzschaltbild der Zellmembran

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

$$I_C = \frac{dq}{dt} = C_m \frac{dU}{dt}. \quad (3.3)$$

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

Gleichungen 3.2 und 3.3 eingesetzt in Gleichung 3.1 ergibt

$$I = \frac{U - U_{Leak}}{R} + C_m \frac{dU}{dt}. \quad (3.4)$$

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

$$RC_m \frac{dU}{dt} = (U_{Leak} - U) + RI. \quad (3.5)$$

Der Strom  $I$  wird als Input-Größe verstanden und im weiteren Verlauf als  $I_{in}$  betitelt. Nach Division durch  $RC$  und Einführung des Leitwerts  $G_{Leak} = \frac{1}{R}$  entsteht die gewollte Form:

$$\frac{dU}{dt} = \frac{G_{Leak}(U_{Leak} - U) + I_{in}}{C_m}. \quad (3.6)$$

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

$$I_{in} = \sum_{i=1}^n I_{Stimuli} + \sum_{i=1}^n I_{Syn} + \sum_{i=1}^n I_{Gap}. \quad (3.7)$$

Die Implementierung dieser Gleichungen und den entsprechenden numerischen Lösungsverfahren findet sich in 3.2. Ein beispielhafter Spannungsverlauf bei einem konstanten, positiv einfließendem Strom  $I_{in}$  wird in Abb. 3.2 dargestellt. Die anliegenden Synapsenströme sind durch folgenden formularen Zusammenhang zu berechnen [7]:

$$I_{Syn} = \frac{w}{1 + e^{\sigma(u_{pre}(t) + \mu)}} (E - u_{post}(t)). \quad (3.8)$$

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 die Kreisfrequenz



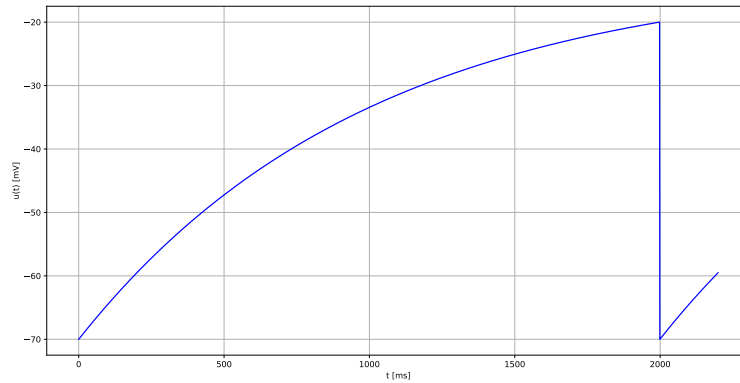


Abb. 3.2: Grafische Darstellung des Membranpotentials durch das Leaky Integrate and Fire - Modell

$w$ , die Standardabweichung  $\sigma$  und der Erwartungswert  $\mu$ .

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

$$I_{Gap} = \hat{w}(u_{post}(t) - u_{pre}(t)). \quad (3.9)$$

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

### 3.2 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 Rezeptoren sind in der Lage, äußere Einflüsse wie bspw. Licht- oder Berührungsintensität in eine 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. 2.3 [7]) 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  $\varphi$ ), *PVD* und *ALM* bedienen eine sekundäre Eingangsgröße (inverses Pendel: Winkelgeschwindigkeit  $\dot{\varphi}$  oder Cartposition  $x$ ). Diese Wahl beruht auf der internen Verschaltung des Netzwerks durch Synapsen und Gap-Junctions.

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:

$$S_{positiv} := \begin{cases} -70mV & x \leq 0 \\ -70mV + \frac{50mV}{x_{min}}x & 0 < x \leq x_{min} \\ -20mV & x > x_{max} \end{cases} \quad (3.10)$$

$$S_{negativ} := \begin{cases} -70mV & x \geq 0 \\ -70mV + \frac{50mV}{x_{min}}x & 0 > x \geq x_{min} \\ -20mV & x < x_{max} \end{cases} \quad (3.11)$$

$x \in [x_{min}, x_{max}]$  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 oben genannten Simulationsumgebung des inversen Pendels entspricht der Ausgang des Netzwerks entweder einer diskreten Vorwärts- oder einer Rückwärtsbewegung. Genauer zu der Interaktion mit dem genannten Simulationskonstrukt im Kapitel 5.

### 3.3 Zuverlässigkeit und Limitationen

Das Leaky Integrate and Fire - Modell ist stark vereinfacht und zeigt die grundsätzlichen Eigenschaften des Membranpotentials auf. Es erfolgt ein lineares Auf-integrieren der anliegenden Ströme und eine simple Rücksetzung des Aktionspotentials nach Überschreitung des Thresholds  $\vartheta$  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.

Da das biologische neuronale Netzwerk qualitativ betrachtet werden soll und eine spätere Optimierung durch Parameterfindung und Gewichtung erfolgt, wird das Leaky Integrate and Fire Modell weiterhin verwendet.

### 3.4 Implementierung

Zur Implementierung des Leaky Integrate and Fire - Modells wird die Programmiersprache Python verwendet. Angelehnt an die Formeln aus Sektion 3.1 kann ein einfacher Algorithmus implementiert werden. Der gesamte Code befindet sich auf der zugehörigen GitHub Repository<sup>1</sup>.

<sup>1</sup> <https://github.com/J0nasW/BA>

Da sich in der Berechnung der Membranpotentiale eine lineare Differenzialgleichung erster Ordnung ergibt (Siehe (3.6)), muss diese entsprechend numerisch gelöst werden. Die Lösung kann durch das Euler-Verfahren sowie durch die Methode nach Runge-Kutta gefunden werden, wobei letztere (4. Ordnung) deutlich genauer ist. Folgende numerische Lösungsverfahren wurden dem Buch „Nonlinear Dynamics and Chaos“ [13] entnommen.

**Anmerkung 3.1 (Numerisches Lösungsverfahren nach Euler).**

Gegeben sei eine Differenzialgleichung 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  $\Delta t$  bekannt sein sowie die Anzahl der Zeitschritte  $T$ . Somit lässt sich die Differenzialgleichung numerisch lösen:

$$x_{n+1} = x_n + f(x_n)\Delta t \quad (3.12)$$

**Anmerkung 3.2 (Numerisches Lösungsverfahren nach erweiterter Euler-Methode).**

Gegeben sei ebenfalls eine Differenzialgleichung 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  $\Delta t$  bekannt sein sowie die Anzahl der Zeitschritte  $T$ . Somit lässt sich die Differenzialgleichung numerisch lösen:

$$\tilde{x}_{n+1} = x_n + f(x_n)\Delta t \quad (3.13)$$

$$x_{n+1} = x_n + \frac{1}{2}[f(x_n) + f(\tilde{x}_{n+1})]\Delta t \quad (3.14)$$

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

**Anmerkung 3.3 (Numerisches Lösungsverfahren nach Runge-Kutta 4. Ordnung).**

Gegeben sei eine Differenzialgleichung 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  $\Delta t$  bekannt sein sowie die Anzahl der Zeitschritte  $T$ . Somit lässt sich die Differenzialgleichung numerisch lösen:

$$\begin{aligned} k_1 &= f(x_n)\Delta t \\ k_2 &= f(x_n + \frac{1}{2}k_1)\Delta t \\ k_3 &= f(x_n + \frac{1}{2}k_2)\Delta t \\ k_4 &= f(x_n + k_3)\Delta t \end{aligned} \quad (3.15)$$

$$x_{n+1} = x_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \quad (3.16)$$

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.

Anwendung der Anmerkung 2.3 auf die Funktion (3.6) resultiert in folgende Berechnung, welche direkt implementiert werden kann:

$$\begin{aligned}
k_1 &= \frac{G_{leak}(U_{leak} - u_i(t)) + (I_{stimuli} + I_{syn} + I_{gap}) \Delta t}{C_m} \\
k_2 &= \frac{G_{leak}(U_{leak} - (u_i(t) + \frac{1}{2}k_1)) + (I_{stimuli} + I_{syn} + I_{gap}) \Delta t}{C_m} \\
k_3 &= \frac{G_{leak}(U_{leak} - (u_i(t) + \frac{1}{2}k_2)) + (I_{stimuli} + I_{syn} + I_{gap}) \Delta t}{C_m} \\
k_4 &= \frac{G_{leak}(U_{leak} - (u_i(t) + k_3)) + (I_{stimuli} + I_{syn} + I_{gap}) \Delta t}{C_m}
\end{aligned} \tag{3.17}$$

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

$$u_{i+1}(t) = u_i(t) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4). \tag{3.18}$$

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:

---

#### Algorithmus 1: compute

---

```

Input :  $x, u, A, B, C_m, G_{Leak}, U_{Leak}, \sigma, w, \hat{w}$ 
Output:  $x, u, fire, I_{syn}, I_{gap}$ 
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    $x[i], fire[i] = U_{neuron}.calc(x[i], I_{inter}[i], I_{sensor}[i], C_m[0, i], G_{Leak}[0, i], U_{Leak}[0, i], v, \Delta t)$ 
27 end
28 return  $x, u, fire, I_{syn}, I_{gap}$ 

```

---

Die Vektoren  $x$  und  $u$  spiegeln die aktuellen Membranpotentiale der jeweiligen Nervenzellen wider:

$$\mathbf{x} = (AVA \ AVD \ PVC \ AVB) \text{ und} \quad (3.19)$$

$$\mathbf{u} = (PVD \ PLM \ AVM \ ALM). \quad (3.20)$$

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 (3.10) und (3.11) je nach Sensordaten gesetzt werden. Matrizen  $\mathbf{A}$  und  $\mathbf{B}$  werden als Transitionsmatrizen genutzt, um die Verbindungen der Nervenzellen im neuronalen Netz zu beschreiben.

$$\mathbf{A} = \begin{matrix} \begin{matrix} \uparrow \\ AVA \\ AVD \\ PVC \\ AVB \end{matrix} & \begin{matrix} AVA & AVD & PVC & AVB \end{matrix} \\ \begin{matrix} AVA \\ AVD \\ PVC \\ AVB \end{matrix} & \begin{pmatrix} 0 & 0 & 1 & 1 \\ 2 & 0 & 2 & 0 \\ 0 & 2 & 0 & 2 \\ 1 & 1 & 0 & 0 \end{pmatrix} \end{matrix} \quad (3.21)$$

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

$$\mathbf{B} = \begin{matrix} \begin{matrix} \uparrow \\ PVD \\ PLM \\ AVM \\ ALM \end{matrix} & \begin{matrix} AVA & AVD & PVC & AVB \end{matrix} \\ \begin{matrix} PVD \\ PLM \\ AVM \\ ALM \end{matrix} & \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 3 & 0 \\ 0 & 3 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} \end{matrix}. \quad (3.22)$$

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 hergestellt. 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  $\mathbf{B}_j$ .

Durch diese Vorschriften ist eine Implementierung des gegebenen neuronalen Netzes möglich und die Berechnung essenzieller Größen durch einfache Matrixoperationen zu realisieren.

Ein entsprechendes Modell des neuronalen Netzes lässt sich durch den Zusammenhang

$$\mathbf{X} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} \quad (3.23)$$

beschreiben. Für die spätere Implementierung sind die gezeigten Gleichungen und Systembeschreibungen essenziell und werden im weiteren Verlauf wieder aufgegriffen.



## Kapitel 4

# 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 diese maximieren soll. Durch die immer größer werdende Datenbasis fällt es dem Agenten mit fortgeschrittener Simulation immer leichter, die „richtigen“ Aktionen zu treffen, um die maximale Belohnung zu erhalten.

### 4.1 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 aufwendige 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 [8].

*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 [8].

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

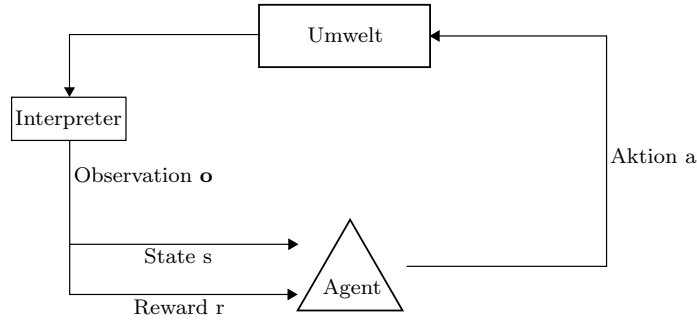


Abb. 4.1: Graphische Darstellung des Reinforcement Learning Algorithmus

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  $\varphi$  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

$$G_t = \sum_{k=0}^T R_{t+k+1} \quad (4.1)$$

Des Weiteren ist es geläufig, einen s.g. „Discount-Faktor“  $\gamma$  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.

$$G_{t\gamma} = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \text{ mit } \gamma \in [0, 1) \quad (4.2)$$



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

## 4.2 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 Potenziale 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 arbeiten wie bereits in Kapitel 2 und 3 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 und Gap-Junction 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. 4.1 stellt die Umwelt unsere Simulationsumgebung des inversen Pendels (**OpenAI Gym - CartPole**v0) dar. Diese nimmt eine Aktion (FWD oder REV) pro Simulationsschritt an und gibt entsprechend einen Observationsvektor  $\mathbf{o}$  aus, welcher durch den Interpretierer übersetzt wird. Der aktuelle State  $s$  wird durch die vier Sensorneuronen *PVD*, *PLM*, *AVM* und *ALM* entsprechend interpretiert und in das neuronale Netz eingegeben.

## 4.3 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 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.

### 4.3.1 Q-Learning

Die Methode des Q-Learnings wurde zuerst durch ein Paper von Watkins [14] definiert und vorgestellt. Voraussetzung um diese Algorithmen anzuwenden ist eine kontrollierte Markov Umgebung.

**Anmerkung 4.1 (Markow Eigenschaft und Umgebung).** Als Markow-Eigenschaft<sup>1</sup> bezeichnet man, wie stark ein stochastischer Prozess von der eigenen Vergangenheit abhängt. Diese Bedingung erlaubt es, Markow-Prozesse zu beschreiben.

Durch eine Markow-Umgebung ist es möglich, aus einer begrenzten Anzahl an vergangenen States die Wahrscheinlichkeit für das Eintreten zukünftiger Ereignisse durch selbst gewählte Aktionen vorherzusagen. [12]

Ziel der Q-Learning Methode ist es, die Qualität der getätigten Aktionen bei unterschiedlichen Simulationsbedingungen zu verbessern. Dabei wird ein Agent eingesetzt, welcher lernt, in einer Markow-Umgebung optimal zu handeln, indem die Konsequenzen der Aktionen sofort zurückgeführt, analysiert und verarbeitet werden. Dem Agenten ist zu jedem Zeitpunkt der Simulation die Umwelt unbekannt.

Unmittelbar nach einer getätigten Aktion  $a$  erhält der Agent neben dem State  $x$  (ausgewählte Messgrößen wie bspw. der Winkel des inversen Pendels  $\varphi$ ) den Reward  $R_x(\pi(x))$ . Aus diesen Informationen lässt sich der Wert des erhaltenen States  $x$  berechnen:

$$V^\pi(x) \equiv R_x(\pi(x)) + \gamma \sum_y P_{xy}[\pi(x)] V^\pi(y) \quad (4.3)$$

mit  $\gamma$  als Diskontierungsfaktor des nächsten Rewards und  $P_{xy}$  als Wahrscheinlichkeit der nächsten vorhergesagten Veränderung der Umwelt.

Nach Watkins existiert mindestens eine optimale stationäre Vorgehensweise („Policy“)  $\pi^*$  für welche gilt

$$V^*(x) \equiv V^{\pi^*}(x) = \max_a \left\{ R_x(a) + \gamma \sum_y P_{xy}[a] V^{\pi^*}(y) \right\}. \quad (4.4)$$

Ziel des s.g. „Q-Learner“ ist es, diese optimale Policy zu finden. So lassen sich die charakteristischen Q-Values

$$Q^\pi(x, a) = R_x(a) + \gamma \sum_y R_{xy}[\pi(x)] V^\pi(y) \quad (4.5)$$

berechnen. Diese spiegeln den erwarteten diskontierten Reward bei Ausführung einer Aktion  $a$  mit State  $x$  und der darauf folgenden Policy  $\pi$  wieder. Zusammenfassend kann durch die Methode des Q-Learning eine optimale Vorgehensweise des Agenten erzielt werden, wenn die entsprechenden Q-Values der optimalen Policy gefunden bzw. erlernt werden.

---

<sup>1</sup> Nach Andrei Markow (1856 - 1922)

### 4.3.2 Gradient Policies

Eine weitere Möglichkeit, Probleme durch Reinforcement Learning zu lösen, ist die Anwendung s.g. „Gradient Policies“. Dies stellt ein klassisches Optimierungsproblem dar und fordert in erster Linie ebenfalls eine Markow-Umgebung.

Es wird eine gegebene Policy  $\pi_\theta(s, a)$  mit Parametern  $\theta$  angenommen. Ziel ist es, die optimalen Parameter  $\theta$  zu finden, um den Reward zu maximieren. Um die Qualität der Policy  $\pi_\theta$  zu messen, wird

$$J(\theta) = V^{\pi_\theta}(s) \quad (4.6)$$

als Qualitätsgröße abhängig von der gegebenen Policy sowie dem State  $s$  eingeführt. Policy Gradient Algorithmen suchen nach einem lokalen Maximum in  $J(\theta)$ , indem sie sich entlang des Gradienten der Policy

$$\Delta\theta = \alpha \nabla_\theta J(\theta) \quad (4.7)$$

bewegen.  $\alpha$  ist dabei ein Schrittweitenparameter. Somit ist  $\nabla_\theta J(\theta)$  definiert als

$$\nabla_\theta J(\theta) = \begin{pmatrix} \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{pmatrix}. \quad (4.8)$$

Bei Anwendung von Gradient Policies stellt sich eine gute Konvergenzeigenschaft des Lernalgorithmus ein. Durch das stetige Bewegen auf dem erlernten Gradienten der Qualitätsgröße  $J(\theta)$  wird ein Maximum gefunden. Jedoch kann dieses nur ein lokales Maximum darstellen, und global gesehen nicht das Optimum widerspiegeln. Darüber hinaus bietet sich die Methode bei großen Aktionsräumen und langen Laufzeiten an, da selbst stochastische Prozesse erlernt werden können. [12]

### 4.3.3 Genetische Algorithmen

Die Anwendung genetischer Algorithmen im Bereich des Reinforcement Learning ist ebenfalls schon seit einiger Zeit bekannt und führt in den richtigen Situationen zu guten Ergebnissen.

Der Grundstein dieser Optimierungsverfahren wurde von Holland et al. in seinem Werk „Adaption in Natural and Artificial Systems“ [5] gelegt. Basierend auf den bereits von Darwin<sup>2</sup> beobachteten Phänomenen der Natur, setzen genetische Algorithmen bei dem Ansatz „Survival of the Fittest“ an. Grundsätzlich kann die Vorgehensweise genetischer Algorithmen wie folgt dargestellt werden [2]:

- Die erste Generation an Parametern wird zufällig initialisiert. Gleich der RandomSearch Methode werden über eine Gleichverteilung verschiedene Parameter erzeugt
- Es werden Simulationen mit den Parametern der ersten Generation gefahren. Die entsprechende Güte wird anhand des Rewards festgelegt

---

<sup>2</sup> Charles Darwin (1809 - 1882)

- Durch eine festgelegte Grenze werden Kandidaten der ersten Generation mit sehr guten Parametern selektiert, welche zur Rekombination genutzt werden
- Die Rekombination kann als eine Aktualisierung der Parametergrenzen verstanden werden.
- Durch Mutation werden anhand der neuen Parametergrenzen wieder zufällige Parameter erzeugt. Diese werden wieder durch Simulation evaluiert und sollten in der Theorie nun bessere Ergebnisse erzielen.
- Wieder erfolgt eine Selektion basierend auf neuen Auswahlkriterien der Mutationen.

Die Schritte der Selektion, Rekombination, Mutation und Evaluierung werden bis zu einem gewählten Abbruchkriterium durchlaufen.

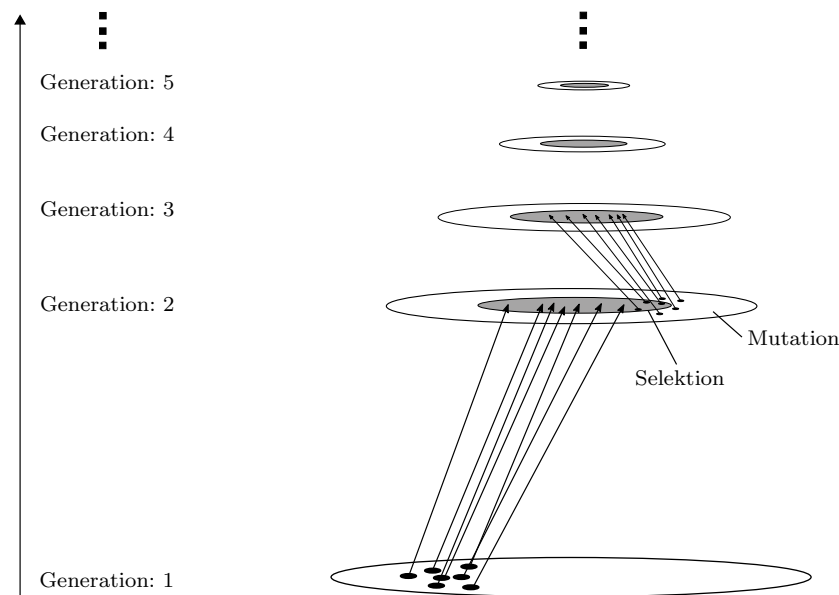


Abb. 4.2: Graphische Darstellung des genetischen Algorithmus

Wie in Abb. 4.2 anschaulich dargestellt, verringert sich mit jeder Generation der Parameterraum und die Simulation konvergiert optimalerweise zu einem globalen Maximum. Dieses hängt jedoch stark von der Anzahl der Selektionen sowie der gewählten Varianz bei Mutation ab. Hier ist ein optimaler Trade-Off zwischen Rechenleistung und Simulationsdauer zu finden.

#### 4.3.4 Random Search

Gewisse Probleme in der Domäne des Reinforcement Learning erfordern einen verallgemeinerten Ansatz zum Finden von optimalen Parametern. Das hier vorliegende Problem bietet eine große Anzahl an lokalen Maxima und bereitet daher den meisten Algorithmen Probleme bei der Anwendung. Ein Anpassen an die geforderten Umstände ist im Grunde möglich, erfordert jedoch

einen hohen Rechenaufwand und führt evtl. zu keiner Verbesserung der Ergebnisse. Daher wird die Methode des Random Search angewendet.

Ähnlich zum genetischen Algorithmus werden Parameter für die Simulation in vorher festgelegten Grenzen durch eine Gleichverteilung erzeugt. Diese werden auf eine Simulation angewendet und die Performance wird anhand des Rewards ausgewertet. Durch ein simples High Score System werden Parameter mit gutem Reward gespeichert, schlechte Simulationen werden verworfen. Durch den Einsatz von genügend Rechenleistung und langen Simulationszeiten können so sehr gute Parameter gefunden werden (siehe Appendix C).

Wie in der vorherigen Sektion 4.2 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	$E_{Excitatory}$	pos. Beeinflussung des Membranpotentials	$[0mV]$
Synapsenstrom	$E_{Inhibitory}$	neg. Beeinflussung des Membranpotentials	$[-90mV]$
Synapsenstrom	$\mu$	Erwartungswert (Modelle)	$[-40mV]$
Synapsenstrom	$\sigma$	Standardabweichung (Modell)	$[0.05, 0.5]$
Synapsenstrom	$w$	Kreisfrequenz der Synapsen	$[0S, 3S]$
Synapsenstrom	$\hat{w}$	Kreisfrequenz der Gap-Junctions	$[0S, 3S]$

Tabelle 4.1: Grenzen der essentiellen Parameter im biologischen neuronalen Netz.

Die gegebenen Grenzen folgen aus [7] und [4] und sind durch Calcium und Potassiummengen im Nervensystem des *C. Elegans* verbunden.



## Kapitel 5

# 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-Repository<sup>1</sup> zu finden. Ein entsprechendes Deckblatt mit weiteren Informationen liegt dieser Arbeit in Anhang C bei.



Abb. 5.1: QR-Code Verweis zur GitHub-Repository.

### 5.1 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.

---

<sup>1</sup> <https://github.com/J0nasW/BA>

Datei: `visualize.py`

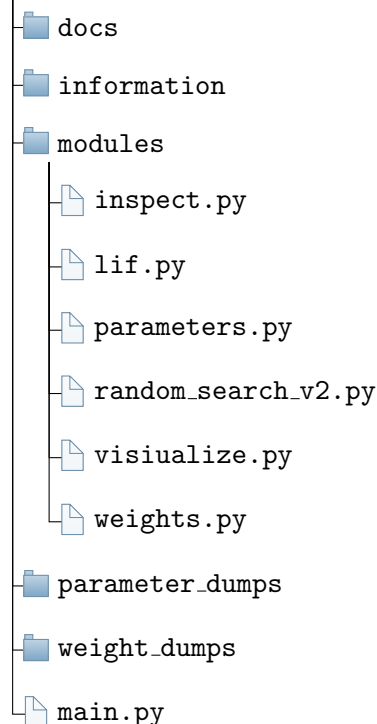
- Es muss die Möglichkeit bestehen, aus bereits existierenden Suchalgorithmen neue Implementationen zu erstellen und gegebene Funktionen einfach einsetzen zu können.

Vorlage: `random_search_v2.py`, `weights.nn.py`

Aufgrund der hohen Vielfältigkeit an neuronalen Netzen und Suchalgorithmen soll dieses Programm als Basis für neue Projekte und Ideen dienen. Daher wird empfohlen, eine Fork des Repositories (siehe Anhang C) 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  $\text{\LaTeX}$ -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, ...) in 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 [3] gespeichert.

## 5.2 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.



### 5.2.1 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 gleich-verteilte, zufällige Werte zu generieren, wird die Funktion `random.uniform` aus dem bekannten Package `numpy` [9] verwendet. Nach Aufruf des Algorith-

---

#### 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**  $C_m, G_{Leak}, U_{Leak}, \sigma, w, \hat{w}$

---

mus `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 sie erfolgreich, andernfalls wird nach Ablauf der Simulationszeit der Algorithmus unterbrochen.

Der gesamte Programmablauf gestaltet sich vereinfacht wie folgt:

---

#### 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  $\leftarrow$  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. 5.3  
 8     **if** `reward  $\geq$  best_reward` **then**  
 9         `Set best_reward  $\leftarrow$  reward`  
 10         `Result = [ $C_m, G_{Leak}, U_{Leak}, \sigma, w, \hat{w}$ ];` // Für Parameter-Dump  
 11         **if** `reward  $\geq$  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`

---

Die elementare Funktion `run_episode` wird in Sektion 5.3 genauer erläutert. Sie enthält unter anderem auch die Funktion `compute`, welche in Sektion 3.4 bereits detailliert thematisiert wurde.

### 5.2.2 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. 2 vorgestellte symmetrische Neuronale Netz (Abb. 2.4) 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.

### 5.2.3 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 Differenzialgleichungen (siehe Sektion 3.4). 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. 2.4 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
$\sigma$	Synapse	16
$w$	Synapse	16
$\hat{w}$	Synapse	2
Gesamt:		<b>46</b>

Somit werden in jedem Simulationslauf zuerst 46 Parameter in gegebenen Grenzen durch eine Gleichverteilung erzeugt und anschließend durch das `compute`-Modul 3.4 die benötigten Synap-

senströme und Membranpotentiale errechnet.

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

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

Somit werden insgesamt 18 Gewichte pro Episode erzeugt und auf das Modell angewendet. Zur Berechnung der erforderlichen Ströme und Potenziale 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<sup>2</sup>, Microsoft mit der Azure Cloud<sup>3</sup> und Google mit der Google Cloud Platform (GCP)<sup>4</sup> 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 Plattform 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<sup>®</sup> 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 auf das System geklont und die benötigten Abhängigkeiten (siehe Anhang B) 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.

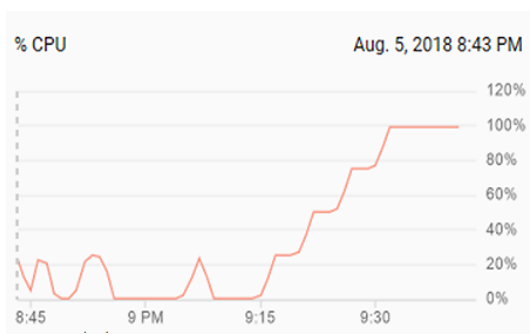


Abb. 5.2: Systemauslastung der virtuellen Instanz - GCP Dashboard

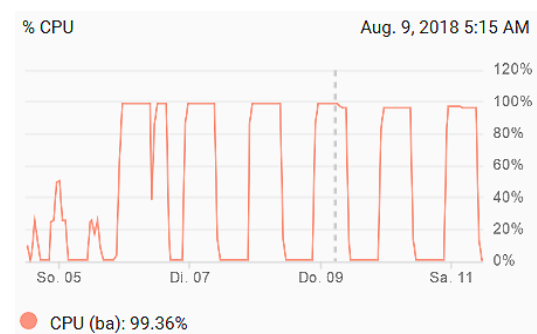


Abb. 5.3: Systemauslastung der virtuellen Instanz im Dauerbetrieb (Cronjob)

Wie in Abb. 5.2 und 5.3 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

<sup>2</sup> <https://aws.amazon.com/de/>

<sup>3</sup> <https://azure.microsoft.com/de-de/>

<sup>4</sup> <https://cloud.google.com/>

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 [3] 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.x visualisiert werden können.

### 5.3 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 OpenAI<sup>5</sup>. OpenAI ist eine Non-Profit Organisation mit dem Ziel, den Forschungsbereich der künstlichen 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

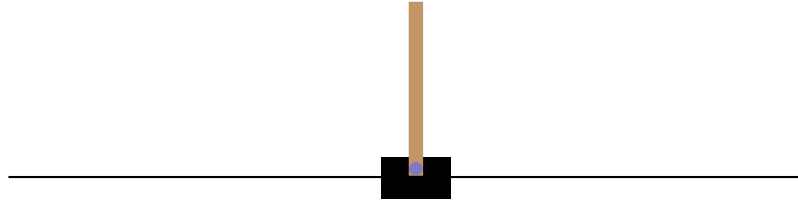
- Robotics

Komplexe Simulationen von Roboterarmen oder -händen

In dieser Arbeit wird die Umgebung `CartPole_v0` (Abb. 5.4) gewählt. Diese besteht aus einem einfachen inversen Pendel, welches sich auf einer zweidimensionalen Bahn frei bewegen kann.

---

<sup>5</sup> <https://openai.com/>

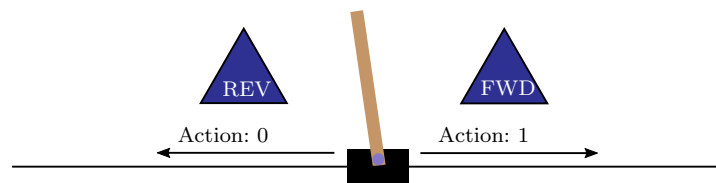
Abb. 5.4: Simulationsumgebung `CartPole_v0`.

Die Handhabung dieser Simulationsumgebung ist dank einer großen Community und guten 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

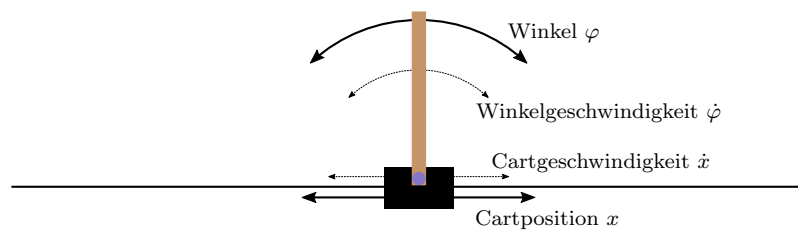
Das bereits vorgestellte neuronale Netz (Abb. 2.4) verfügt über zwei Motor-Neuronen, welche als Eingang der Simulation genutzt werden können. Interne Nervenzellen *AVA* und *AVB* sind durch eine direkte Synapse mit den genannten Motor-Neuronen verbunden und verursachen im Falle eines Fire-Events die Bewegung des Wagens um einen Schritt nach links bzw. rechts. Pro Simulationsschritt erfolgt je nach Observation immer ein Fire-Event, sodass das Pendel zu jeder Zeit eine berechnete Aktion erhält.

Abb. 5.5: Simulationsumgebung `CartPole_v0` mit Aktionen *FWD* und *REV*.

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

$$\mathbf{o} = (C_{Position} \ C_{Velocity} \ P_{Angle} \ P_{Velocity}). \quad (5.1)$$

<i>Parameter</i>	<i>Beschreibung</i>	<i>Grenzen</i>
$C_{Position}$	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. 5.6: Simulationsumgebung `CartPole_v0` mit Observationsgrößen.

Die Informationen aus dem Observationsvektor  $\mathbf{o}$  werden entsprechend interpretiert und den Sensor-Neuronen zugeführt. Wie bereits in Sektion 2.3 beschrieben, ist der Winkel des Pendels  $\varphi$  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 ist dem neuronalen Netz zuzuschreiben. Durch die inhibitorische Beeinflussung durch Sensor-Neuronen *ALM* und *PVD* entsteht ein dämpfendes Signal, welches dafür sorgt, dass bei zu hoher Winkelgeschwindigkeit oder Cartposition die Kompensation durch Bewegung des Carts nicht übersteuert. Ergebnisse durch Wahl der geeigneten sekundären Sensorgröße werden in Kapitel 6 erläutert.

## 5.4 Visualisierung und Auswertung

Wie bereits in Sektion 5.3 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` [6] 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.

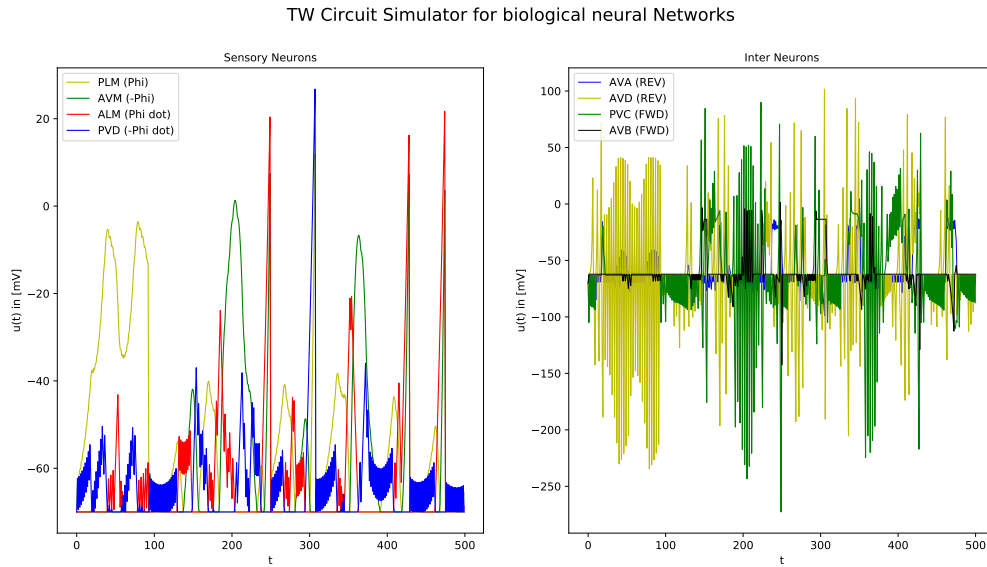


Abb. 5.7: Plot der Membranpotentiale von Inter- und Sensorneuronen

Weiterhin wird standardmäßig neben den Membranpotentialen auch der anliegende Synapsenstrom an jeder Nervenzelle geplottet. Diese Veranschaulichung gibt Aufschluss über die gewählten Parameter und das Feuerverhalten im internen neuronalen Netzwerk.

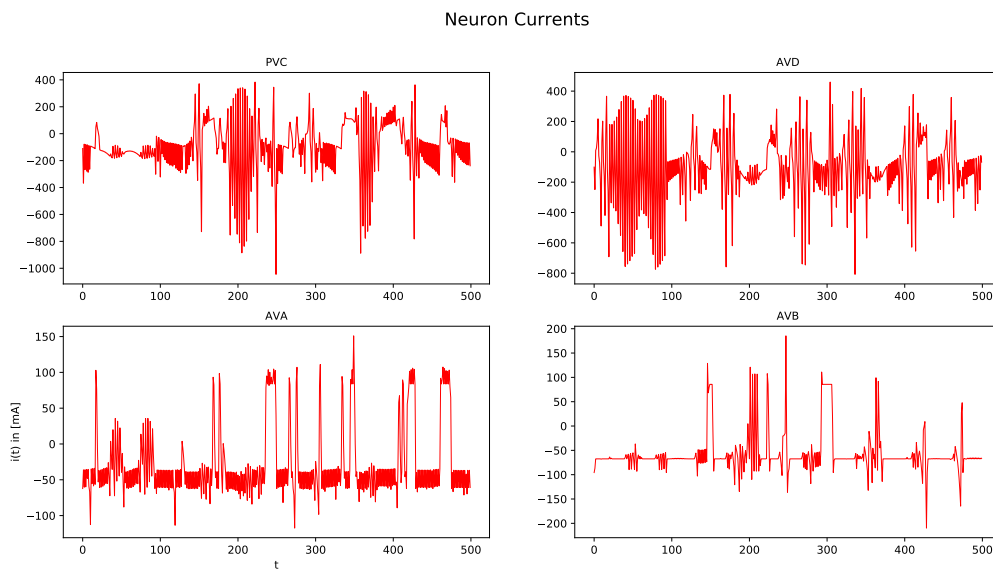


Abb. 5.8: Plot der anliegenden Ströme aus Stimulus, Synapsen und Gap-Junctions

## 5.5 Sonstige Implementierung

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

### 5.5.1 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 Parameter zur Berechnung von Synapsenströmen und Membranpotentialen. Alle weiteren Daten werden automatisch berechnet und im Laufe der Simulation bereitgestellt.

### 5.5.2 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` Dateiendung 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 unter unterschiedlichen 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) [3] gespeichert. Dies ist zum einen performanter, sorgt zum anderen für eine erheblich größere Kompatibilität unter Python-Versionen und Betriebssystemen.

### 5.5.3 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 Konsolen-Output erfolgen oder ein Plot der gespeicherten Daten erzeugt werden.



## Kapitel 6

# Performance & Auswertung

Dieses Kapitel dient der Auswertung von Versuchsergebnissen und dem Vergleich der verschiedenen Suchalgorithmen. Ausgangspunkt und somit Vergleichskriterium sind Simulationsgrößen des inversen Pendels `CartPole.v0` des Frameworks OpenAI Gym. Es wurde zuerst eine Berechnungsgrundlage eines neuronalen Netzes nach C. Elegans [1] geschaffen und implementiert. Dazu wurden verschiedene numerische Lösungsverfahren von Differenzialgleichungen verglichen und umgesetzt [13]. Letztlich wird ein universeller Simulator geschaffen, welcher Informationen über die Nervenzellen und Synapsen erhält und entsprechend in der Lage ist, das 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 RandomSearch konzentriert sowie auf die Optimierungsmethode durch Gewichten der entsprechenden Synapsen.

### 6.1 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 ressourceneffizient implementiert wurden.

Anfangen bei den Berechnungsmodulen in der Datei `lif.py` wird für komplexere mathematische Operationen die Erweiterung `NumPy` [9] aus dem bekannten Python-Paket `SciPy` [9] 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` [6] oder `hickle` [3] 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 der Optimierung durch den Algorithmus `Weights` ausschlaggebend. Diese Algorithmen wurden im Laufe der Implementierung immer wieder optimiert und verbessert, sodass eine zuverlässige Simulation mit effizien-

ten Laufzeiten möglich wird. Bei festen Simulationszeiten werden auf der bereits vorgestellten virtuellen Instanz folgende Ergebnisse erzielt (Stichprobenartig aufgelistet):

<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	<b>200</b>	12 Std.	10.852.326

Tabelle 6.1: 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	<b>200</b>	12 Std.	3.686.723

Tabelle 6.2: Optimierung durch Algorithmus `Weights`.

Diese Simulationen wurden ausnahmslos auf derselben virtuellen Instanz (parallel) ausgeführt. Die genauen Spezifikationen wurden in Sektion 5.2 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 ca. 11 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` das 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 C genauer erläutert und veranschaulicht.

## 6.2 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.

### 6.2.1 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 aus, wird diese Simulation verworfen. So baut sich ein High-Score-System auf und nach Ablauf der Simulationszeit werden die Parameter mit dem höchsten erreichten Reward gespeichert. Wie darüber hinaus in Abb. A.1 noch einmal verdeutlicht, werden gute Parameter für stabile Simulationsläufe durch den simplen Input des Rewards gefunden.

Nach Anwendung der Parametersuche durch **RandomSearch** wird eine Optimierung des neuronalen Netzes durch den Optimierungsalgorithmus **Weights** durchgeführt. Durch das Einführen von Gewichten für Synapsen und Gap-Junctions, können gewisse Informationsbahnen eingestellt und die Simulation weiter verbessert werden. In Abb. A.2 wird der gesamte Programmablauf noch einmal verdeutlicht. Gefundene Parameter und Gewichte guter Simulationsläufe werden in Appendix C aufgeführt.

Aufgrund der starken Symmetrie des gegebenen Problems und des neuronalen Netzes ist es darüber hinaus möglich, eine symmetrische Parameter- und Gewichtsgenerierung zu implementieren. Anstatt der gesamten 46 Parameter für Nervenzellen Synapsen und Gap-Junctions, werden jeweils nur die Hälfte der benötigten Parameter generiert und anschließend 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.

### 6.2.2 Alternative Such- und Optimierungsalgorithmen

Wie bereits in Sektion 4.3 vorgestellt, existieren 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.

Der in Sektion 4.3.1 zuerst vorgestellte Algorithmus des Q-Learning wurde speziell für die neuen Praktiken der künstlich erschaffenen neuronalen Netze entwickelt. Hier wird auf die Gewichtung vieler einfacher Verbindungen zwischen Neuronen unterschiedlicher Ebenen fokussiert. Die Grundlagen könnten auf einfache Parameteroptimierungen angewendet werden, jedoch ist diese Richtung größtenteils unerforscht.

Die Methode, durch Gradient Policies eine Parameteroptimierung durchzuführen, ist durchaus möglich und kann zu effizienteren Simulationslaufzeiten führen. Problematisch ist hier die Tatsache, dass sich durch die Anzahl an zu simulierenden Parametern viele lokale Maxima in der Reward-Funktion bilden und somit der Algorithmus schnell zu einem Ende kommt. Die grundsätzliche Vorgehensweise beginnt analog zu **RandomSearch** mit einer zufälligen Generierung von Parametern und einem ersten Simulationslauf. Nach der ersten Episode wird ein weiterer zufälliger Parametersatz generiert und eine zweite Simulation gefahren. Ist der Reward dieser zweiten Episode höher, werden die Parameter entsprechend verglichen und die Grenzen zur Generierung neuer, zufälliger Parameter für die nächsten Episoden aktualisiert.

Die dritte Herangehensweise der genetischen Algorithmen stellt sich als robustere Methode zu Gradient Policies heraus. Durch die Wahl mehrerer Simulationsläufe mit guten Rewards und die Expansion dieser Selektion (als Mutation) durch Varianzen wird ebenfalls zielgerichteter gesucht und die Chance, ein globales Optimum zu finden, erhöht. Die Implementierung dieser Methode warf jedoch einige Hürden auf und führte letztendlich zu geringeren Rewards als RandomSearch. Einzig die Optimierung durch Gewichtung von Synapsen und Gap-Junctions kann mit bekannten Algorithmen und den Input des Rewards das Ergebnis effizient beeinflussen.

### 6.3 Zusammenfassung

In Zeiten neuer Denkweisen und unkonventioneller Herangehensweisen an bestehenden Problemen findet die Anwendung neuronaler Netze und künstlicher Intelligenzen immer mehr Anklang in fast allen Bereichen des Alltags sowie der Wissenschaft. Neue Theorien und Lernalgorithmen lassen hoch parallele Rechenkonstrukte lernen und nutzen diese Methoden, um komplexe Aufgabenstellungen zu lösen. Diese Arbeit untersucht zum einen die Grundzüge des Deep Learning mit Schwerpunkt auf Reinforcement Learning, wendet diese Erkenntnisse jedoch auf unkonventionellem Wege auf bereits bestehende neuronale Netze echter Lebensformen an. Dieser Ansatz wurde noch nicht oft untersucht und bietet eine neue Möglichkeit, von der Evolution zu lernen und besser als bereits bestehende Systeme zu werden.

Dies erfordert jedoch zum einen ein sehr genaues und umfassendes Verständnis von biologischen und chemischen Prozessen in neuronalen Netzen sowie medizinische Grundlagen, zum anderen einen ausgereiften Simulator, welcher diese Prozesse effizient und mit hoher Genauigkeit simulativ nachbilden kann. Darüber hinaus erfordert es gute Lernalgorithmen, um unbekannte Parameter und Assoziationen in neuronalen Netzen zu finden und diese auszubilden.

Prozesse innerhalb neuronaler Netze wurden bereits gut erforscht und dokumentiert. Daher konnte hier auf eine große Wissensdatenbank zurückgegriffen werden, um formulare Zusammenhänge für Größen wie Synapsenströme und Membranpotentiale herzuleiten [16]. Dies ist für den Simulator von großer Bedeutung, da somit die Signalverläufe und Fire-Events dargestellt werden können. Wie anhand der Plots 5.7 und 5.8 zu sehen ist, liefert das Leaky Integrate and Fire Modell eine sehr genaue und zuverlässige Simulation der Nervenzellen. Voraussetzung für eine realitätsnahe Simulation sind jedoch stimmige Parameter in jeder einzelnen Nervenzelle und Synapse bzw. Gap-Junction. Diese Parameter wurden durch Suchalgorithmen und langen Simulationsläufe gefunden und zeigen letztendlich den ursprünglichen Reflex des Wurms C. Elegans: Touch-Withdrawal (zu Deutsch: Rückwärtsbewegung bei Berührung). Weiterhin wurde der Simulator ausgebaut, um ein gegebenes Netzwerk zur Regelung dynamischer Systeme zu nutzen. Auch dies konnte mit Erfolg nachgewiesen werden, wie in Appendix C zu sehen ist.

Zusammenfassend zeigte dieses Thema durch Kombination verschiedener Bereiche der Wissenschaft eine neue Herangehensweise an bereits bestehende Probleme der Regelungstechnik auf. Durch die Aktualität und Neuigkeit der genutzten Werkzeuge und Informationsquellen entstand eine Implementierung, welche durch das Einsetzen des Touch-Withdrawal Neuronal Circuit [7] erfolgreich in der Lage war, dynamische Systeme in der Regelungstechnik zu stabilisieren.

Nichtsdestotrotz gibt es nach wie vor viele Baustellen und Erweiterungsmöglichkeiten. Das Regeln des inversen Pendels ist weiterhin durch kleine Unstimmigkeiten schwierig und bricht in manchen Simulationen ab. Dies ist wahrscheinlich auf die Auswahl der Parameter sowie des Designs der bisher implementierten Suchalgorithmen zurückzuführen. Wie bereits in Sektion

4.3 beschrieben, können durch umfangreiche Anpassungen weitaus effizientere Suchalgorithmen angewendet werden. Auch das Netzwerk an sich kann überdacht und durch erweiterte Literaturrecherche näher der Biologie des Wurms *C. Elegans* nachempfunden werden.

## 6.4 Ausblick

Die Anwendung des Reinforcement Learning oder generell des Deep Learning in Bereichen der Regelungstechnik ist noch sehr neu. Besonders der Ansatz, ein existierendes neuronales Netz zur Steuerung dynamischer Systeme zu verwenden, wurde in Fachkreisen noch nicht sehr oft dokumentiert. Daher basiert diese Arbeit auf sehr viel Grundlagenforschung zu Nervensystemen des Wurms *C. Elegans* [1] sowie der Methoden des Reinforcement Learning [8] [11], um eine Brücke zwischen diesen Themen zu schlagen. Jedoch bietet diese Arbeit gleichzeitig nur ein grundlegendes Verständnis über die Prozesse biologischer neuronaler Netze und dessen Implementierung in einen Simulator.

Schon die Berechnungsmodelle der Synapsenströme und Membranpotentiale sind vereinfacht dargestellt. Wie in dem Buch *Neuronal Dynamics* [16] zu lesen ist, bietet das Leaky Integrate and Fire Modell zwar zeit-genaue und zuverlässige Verhaltenssimulationen der Nervenzellen, hat jedoch an anderen Stellen Nachteile, wie in Sektion 3.3 beschrieben.

Weitergehend sollen Parameter und Gewichte für das neuronale Netz gefunden werden, um eine korrekte Simulation und Regelung zu erhalten. Dies ist bisher nur durch die Methode Random-Search geschehen. Weitere Suchalgorithmen (beschrieben in Sektion 4.3) könnten durch große Anpassungen ebenfalls Anwendung in der Suche nach passenden Parametern finden.

Das Framework des Simulators ist bisher mit dem Projekt gewachsen. Obwohl immer stets auf den universellen und modularen Aufbau geachtet wurde, könnte der Simulator noch allgemeiner implementiert sein und eine größere Anzahl an Funktionen aufweisen. Eine Implementierung als Paket in Python mit Verfügbarkeit über PyPI<sup>1</sup> könnte ebenfalls realisiert werden.

Abschließend wurde durch diese Arbeit ein Grundstein gelegt, welcher grundsätzliche Informationen zu den Aspekten biologischer neuronaler Netze, neuronaler Dynamik, und Algorithmen des Reinforcement Learning bietet. Darüber hinaus kann der Simulator TW Circuit aufgrund des modularen Aufbaus beliebig erweitert werden. Ein Funktionsbeweis liefert die Simulation des inversen Pendels `CartPole.v0` aus dem Paket OpenAI Gym, welche mit Erfolg durch einen Reward von 200 bestanden wurde.

---

<sup>1</sup> [www.pypi.org](http://www.pypi.org) - Python Package Index



## Anhang A

# UML-Diagramme wichtiger Algorithmen

### A.1 RandomSearch

Die folgende Abbildung A.1 stellt den genutzten Algorithmus **RandomSearch** qualitativ in einem Programmablaufplan vor.

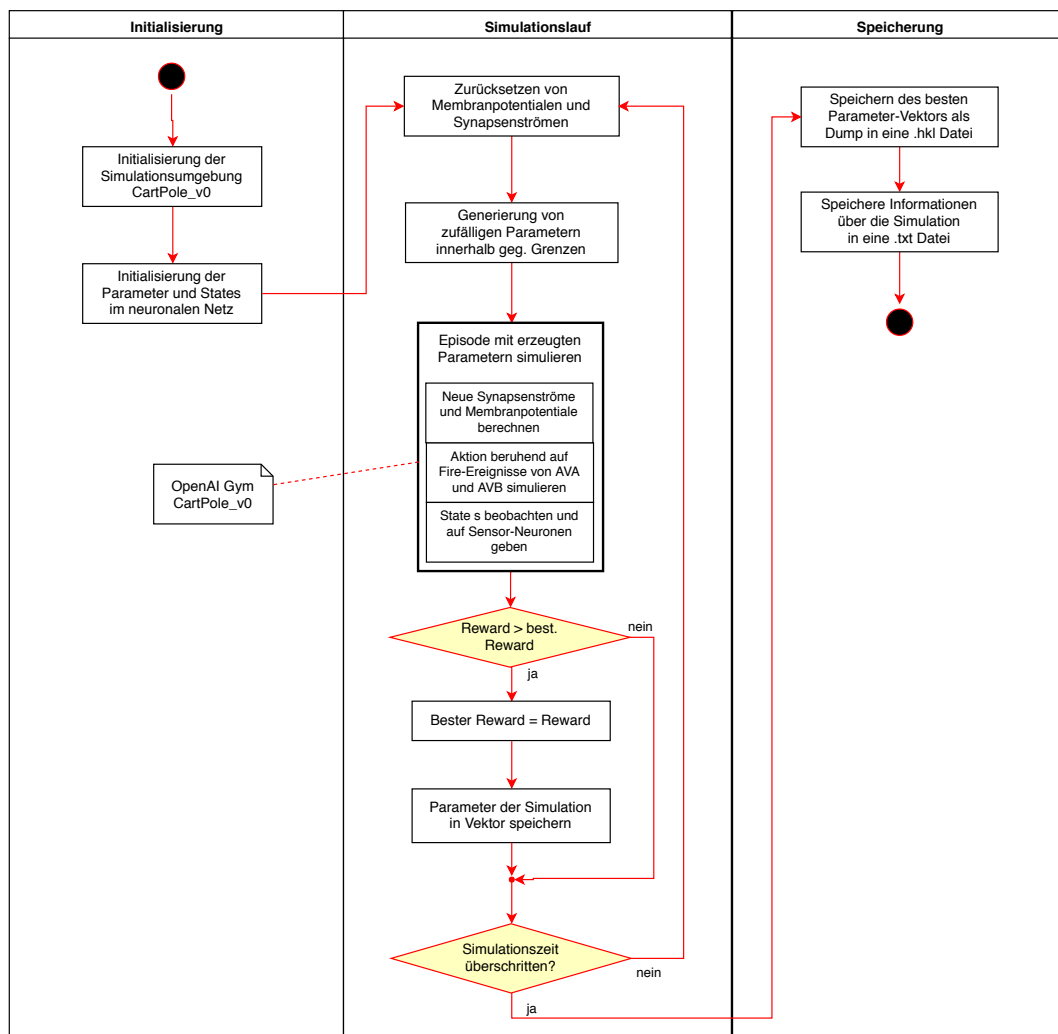


Abb. A.1: UML Diagramm des Algorithmus **RandomSearch**

## A.2 Weights

Die folgende Abbildung A.2 stellt den genutzten Algorithmus **Weights** qualitativ in einem Programmablaufplan vor.

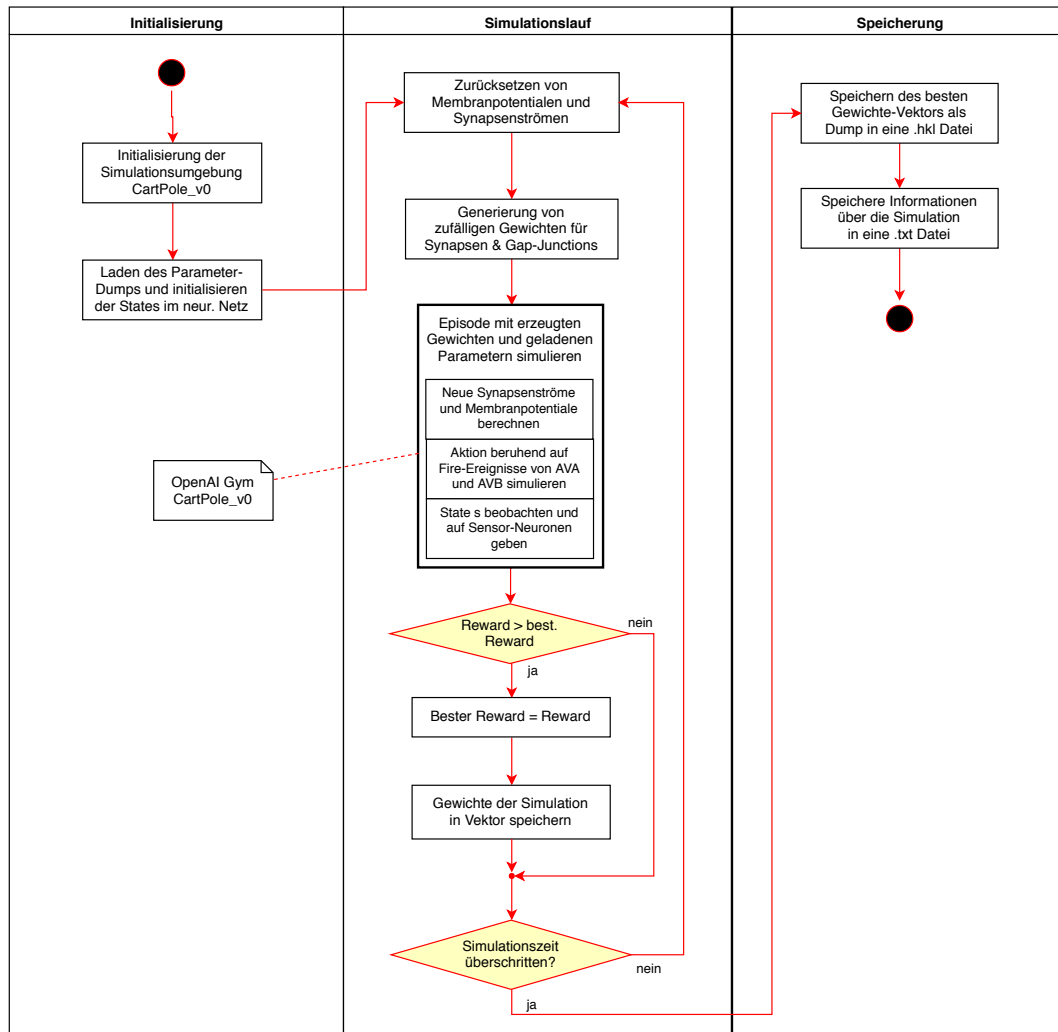


Abb. A.2: UML Diagramm des Algorithmus **Weights**



## Anhang B

# Datenblatt Programm: TW Circuit

Name                       
TW Circuit

## Kurzinfo

Das Programm TW Circuit dient als Simulator des Touch Withdrawal Circuit des Wurms *C. Elegans* [7]. Es kann ein beliebig aufgebautes, neuronales Netz mit biologischem Hintergrund implementiert werden. Dazu wird das Leaky Integrate and Fire Modell genutzt, um die neuronale Dynamik zu simulieren und Fire-Events vorherzusagen. Der Output wird auf eine beliebige Simulationsumgebung gegeben, in diesem Beispiel wird OpenAI's CartPole\_v0 genutzt.

Der Simulator ist OpenSource und kann beliebig erweitert oder verändert werden. Er verfügt darüber hinaus auch über Inspektionsfunktionen sowie gut dokumentiertem Code, um Befehle schnell nachzuvollziehen.

## Dependencies

- 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

## QR-Code



## Dateibaum

### TW Circuit

```

├── docs
├── information
├── modules
│   ├── inspect.py
│   ├── lif.py
│   ├── parameters.py
│   ├── random_search_v2.py
│   ├── visualize.py
│   └── weights.py
├── parameter_dumps
├── weight_dumps
└── main.py
  
```

## Anhang C

### Parameter mit guten Simulationsergebnissen

Nachfolgend werden verschiedene Simulationsläufe mit guten Ergebnissen im Detail in Tabelle C.1 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 werden anhand der Transitionsmatrizen 3.21 und 3.22 in Vektorschreibweise dargestellt:

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

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

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

Gewichte der Synapsen ( $\sigma$ ,  $w$ ) und Gap-Junctions ( $\hat{w}$ ) werden anhand der Transitionsmatrizen 3.21 und 3.22 in Vektorschreibweise dargestellt:

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

Weitere Parameter und Einheiten sind der Tabelle 4.1 zu entnehmen. In Abb. C.1 werden den Synapsen des bekannten neuronalen Netzes Nummerierungen hinzugefügt, um die Parameter der Synapsen und Gap-Junctions zuzuweisen zu können.

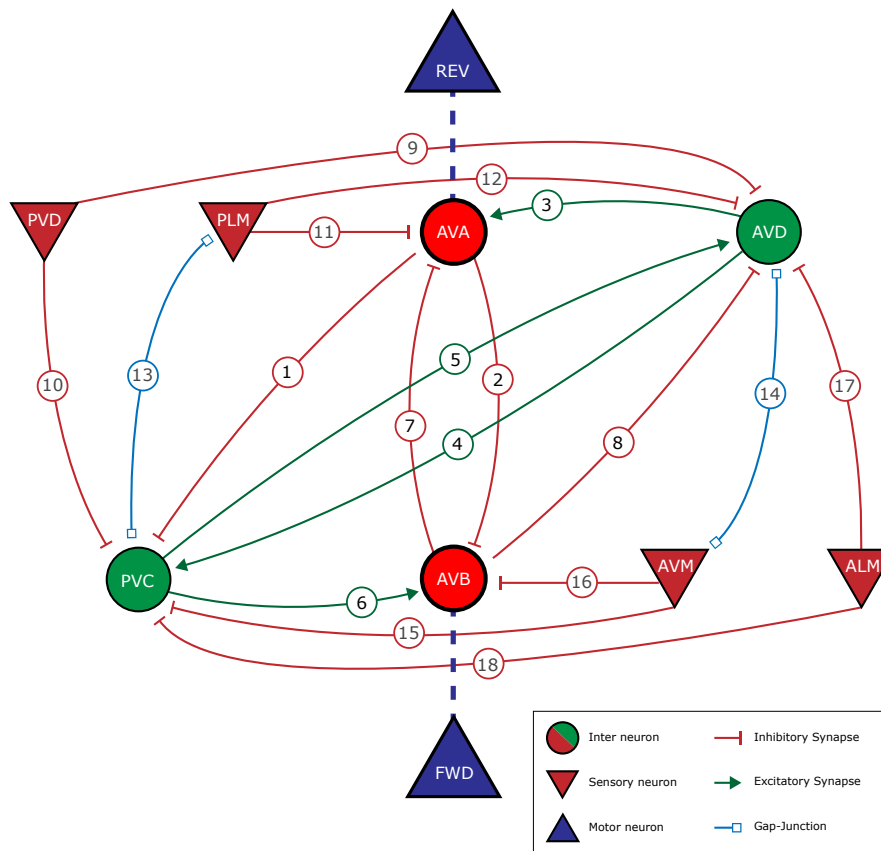


Abb. C.1: Neuronales Netz mit Legende für Gewichte

<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 $\mathbf{C}_m = (0.01, 0.01, 0.01, 0.01),$ $\mathbf{G}_{Leak} = (1.42, 0.63, 1.42, 0.63),$ $\mathbf{U}_{Leak} = (-63.2, -62.2, -63.2, -62.2) .$  Parameter der Synapsen & Gap-Junctions $\boldsymbol{\sigma} = \begin{pmatrix} 0.16, 0.49, 0.15, 0.29, 0.16, 0.49, 0.15, 0.29, \\ 0.13, 0.39, 0.17, 0.06, 0.13, 0.39, 0.17, 0.06 \end{pmatrix},$ $\mathbf{w} = \begin{pmatrix} 0.97, 1.57, 1.48, 1.28, 0.97, 1.57, 1.48, 1.28, \\ 1.71, 1.45, 0.78, 2.84, 1.71, 1.45, 0.78, 2.84 \end{pmatrix},$ $\hat{\mathbf{w}} = (1.84, 1.84).$
20180817_13-56-01	Weights - Gewichte	200	12h, ca. 400.000 Simulationen	Gewichte der Synapsen $\mathbf{g} = \begin{pmatrix} 0.94, 0.96, 0.95, 0.14, 0.94, 0.96, 0.95, 0.14, 0.75, \\ 0.45, 0.54, 0.21, 0.97, 0.75, 0.45, 0.54, 0.21, 0.97 \end{pmatrix}.$

Tabelle C.1: Simulationsläufe mit guten Ergebnissen



# 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. Goldberg DE (1989) Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley Professional, URL <https://www.amazon.com/Genetic-Algorithms-Optimization-Machine-Learning/dp/0201157675?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0201157675>
3. Group TH (2018) Hierarchical data format, version 5. URL <http://www.hdfgroup.org/HDF5/>
4. Hasani RM, Beneder V, Fuchs M, Lung D, Grosu R (????) Sim-ce: An advanced simulink platform for studying the brain of caenorhabditis elegans
5. Holland JH (1992) Adaptation in Natural and Artificial Systems. MIT University Press Group Ltd, URL [https://www.ebook.de/de/product/2864942/john\\_h\\_holland\\_adaptation\\_in\\_natural\\_and\\_artificial\\_systems.html](https://www.ebook.de/de/product/2864942/john_h_holland_adaptation_in_natural_and_artificial_systems.html)
6. Hunter JD (2007) Matplotlib: A 2d graphics environment. Computing in Science & Engineering 9(3):90–95, DOI 10.1109/mcse.2007.55
7. Lechner M, Grosu R, Hasani RM (2017) Worm-level control through search-based reinforcement learning
8. LeCun Y, Bengio Y, Hinton G (2015) Deep learning. Nature 521:436, URL <http://dx.doi.org/10.1038/nature14539>
9. Oliphant TE (2006) A guide to NumPy. Trelgol Publishing USA
10. 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=OJYN1NVW651KCA56C102&tag=teckie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=3540563539>
11. Russell S, Norvig P (2016) Artificial Intelligence: A Modern Approach. Always learning, Pearson Higher Education & Professional Group, URL <https://books.google.de/books?>

id=XS9CjwEACAAJ

12. Silver D (2015) Ucl course on reinforcement learning, URL <http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html>
13. 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>
14. Watkins CJCH, Dayan P (1992) Q-learning. Machine Learning 8(3):279–292, URL <https://doi.org/10.1007/BF00992698>
15. 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>
16. 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