Fakultät für Physik und Astronomie

Ruprecht-Karls-Universität Heidelberg

Bachelorarbeit

im Studiengang Physik

vorgelegt von

Marvin Albert

aus Heidelberg

August 2010

Liquid Computing mit Neuromorpher Hardware

Diese Bachelorarbeit wurde von Marvin Albert ausgeführt am
Kirchhoff-Institut für Physik
unter der Betreuung von
Prof. Dr. Karlheinz Meier

Liquid Computing mit Neuromorpher Hardware
Im Rahmen der vorliegenden Arbeit wurde erstmals das sogenannte Liquid Computing aktionspotenzialbasiert auf einer analog-digitalen, hochkonfigurierbaren und massiv beschleunigten neuromorphen Hardware umgesetzt. Es handelt sich dabei um ein Modell der Informationsverarbeitung, das generische Strukturen, wie z. B. teilweise randomisiert verbundene neuronale Netzwerke, für universelle Berechnungen verwendbar macht. Auf Grund ihrer Beschleunigung und hohen Konfigurierbarkeit eignet sich die in dieser Arbeit verwendete Hardware grundsätzlich optimal für die Realisierung und systematische Untersuchung des Liquid Computing Paradigmas. Die präsentierten Experimente stellen einen Machbarkeitsbeweis dar, welcher ergänzt wird durch darauf aufbauende Studien ausgewählter Parameter. Im Zuge dessen wird auch gezeigt, dass die durchgeführten Klassifizierungen nicht trivial sind und das neuronale Medium in der implementierten Architektur eine Fähigkeit zur Speicherung von Information besitzt. Ein weiterer Kernaspekt der Arbeit sind Beiträge zur Qualitätssicherung der im Rahmen der vorgestellten Experimente verwendeten Ansteuerungssoftware.

Liquid Computing with a Neuromorphic Hardware System
This thesis addresses the first time realization of the so-called Liquid Computing concept on a spike-based mixed-signal, highly configurable and massively accelerated neuromorphic hardware device. Liquid Computing represents an information processing model that enables generic structures, including neural networks with partly randomised connections, to be applied for universal computational purposes. Regarding its high acceleration and configurability, the used hardware system is optimally suited for the implementation and systematic investigation of the Liquid Computing paradigm. Therefore, the experiments presented in this work provide a proof of concept and additionally study the effects of selected parameters. It is shown that the performed classification task is not trivial and that the implemented network architecture possesses the ability to store information. Another key aspect of this work are contributions to the quality assurance of the software framework used to run the described experiments.

Inhaltsverzeichnis

Einleitung

Die Neurowissenschaften studieren die Grundlagen und Mechanismen neuronaler Informationsverarbeitung. Dies beinhaltet die Frage nach der Repräsentation, Verwertung und Speicherung von Information innerhalb von Nervensystemen. Ein entscheidender Ausgangspunkt ist die Beobachtung, dass Organismen von den Informationen über ihre Umwelt sehr effektiv Gebrauch machen können und dabei gleichzeitig höchst anpassungsfähig sind. Das Verständnis der Funktionsweise von Nervensystemen könnte daher unter anderem die Grundlage der Entwicklung neuer Technologien bilden, die sich der Fähigkeit und Flexibilität neuronaler Netze bedient. In dieser Hinsicht hat sich in der Neurowissenschaft ein sehr aktives Forschungsfeld ausgebildet, bei dem sich jedoch bisher keine eindeutige Stoßrichtung etabliert hat. Eine Vielzahl von Methoden und Ansätzen wurde entwickelt, allerdings steht ein zufriedenstellender Zugang zum umfassenden Verständnis neuronaler Informationsverarbeitung noch aus.

Auf Grund der technischen Schwierigkeiten, die bei der Untersuchung lebender Nervensysteme bestehen, stellt die *Modellierung* einen wichtigen Ansatz dar. Modelle können ihren biologischen Originalen unterschiedlich präzise entsprechen und dabei gezielt bestimmte Aspekte aufgreifen oder fallen lassen. Dies ermöglicht Aussagen darüber, welche funktionale Rolle einzelne modellierte Elemente im Kontext eines Netzwerks spielen.

Die Berechnung und Auswertung der Modelle übernehmen meist Computer, indem sie numerische Simulationen von Experimenten an ihnen durchführen. Dies stößt jedoch mit zunehmender Komplexität der Experimente rasch auf Grenzen der Praktikabilität (siehe Kap. 1.1.1). Um dieses Problem in Angriff zu nehmen, wurde im Rahmen des interdisziplinären FACETS[1]-Projekts von der *Electronic Vision(s) Group*[2] eine auf Mixed-Signal[3] VLSI[4]-Technologie aufbauende *neuromorphe Hardware* (siehe Kap. 1.1.2) entwickelt, die programmierbare neuronale Netze emuliert. Diese Bemühungen stellen unter anderem technologische Grundlagenforschung für zukünftige neuronale Computer dar.

Die Aufgabenstellung der hier präsentierten Arbeit besteht darin, ein unkonventionelles Modell der Informationsverarbeitung, das sogenannte *Liquid Computing* (siehe Abschnitt 1.3), zum ersten Mal auf der neuromorphen FACETS-Hardware (siehe Abschnitt 1.1.2) umzusetzen. Liquid Computing verwendet für universelle Berechnungen generische Netzwerke, wobei sich die FACETS-Hardware auf Grund ihrer hohen Konfigurabilität und Emulationsgeschwindigkeit für deren Realisierung und systematische Untersuchung besonders eignet. Aufbauend auf einem Machbarkeitsbeweis soll zu den für die experimentelle Umsetzung nötigen methodischen Grundlagen beigetragen werden.

Eine erfolgreiche vorherige Umsetzung von Liquid Computing auf VLSI Hardware wurde in der Electronic Vision(s) Group, in der auch die vorliegende Arbeit durchgeführt wurde, von Felix Schürmann sowie vom Betreuer dieser Arbeit, Daniel Brüderle, vorgenommen [5, 42]. Dabei wurden, im Gegensatz zur hier präsentierten aktionspotentialbasierten Variante, sogenannte *Perzeptrone* (implementiert im HAGEN[5]-Chip [41]) verwendet, deren Neurone zeitlich diskret und mit binären Ein- und Ausgaben arbeiten. Die Anfänge der im folgenden präsentierten Experimente mit Liquid Computing auf der FACETS-Hardware fanden in enger Kooperation mit Mihai Petrovici, Johannes Bill, Bernhard Vogginger sowie vor allem Sebastian Jeltsch und Daniel Brüderle im Rahmen des Capo Caccia Workshops[6] statt. Die hier präsentierte Umsetzung schließlich entstand aus selbstständiger Arbeit des Autors, profitierte dabei jedoch von den Erfahrungen des

[1] Fast Analog Computing with Emergent Transient States - http://facets.kip.uni-heidelberg.de/
[2] *Kirchhoff-Institut für Physik, Universität Heidelberg*
[3] Gemeinsame Verarbeitung analoger und digitaler Signale
[4] Very Large Scale Integration, d.h. Tausende bis Millionen von Transistoren auf einem einzigen Chip
[5] Heidelberg Analog Evolvable Network
[6] The 2010 CapoCaccia Cognitive Neuromorphic Engineering Workshop

genannten Pilotprojektes.

Im folgenden Kapitel 1 wird eine kurze Einführung in den wissenschaftlichen Kontext der neuromorphen Modellierung und des Liquid Computing im Besonderen gegeben. Zur Abbildung gegebener Netzwerkarchitekturen auf die FACETS-Hardware wird ein komplexes Softwaresystem (siehe Kap. 1.1.4) verwendet, dessen Qualitätssicherung von fundamentaler Wichtigkeit für eine wissenschaftlich verwertbare Durchführung und Auswertung von Experimenten ist. Kapitel 2 beschreibt die zur Qualitätssicherung im Rahmen dieser Arbeit geleisteten Beiträge. Die tatsächliche Hardware-Implementierung des Liquid Computing wird in Kapitel 3 behandelt. Es werden Studien exemplarischen Charakters präsentiert, welche die Eignung von Liquid Computing für die Realisierung auf Hardware belegen. Abschließend werden in Abschnitt 4 eine Zusammenfassung der gewonnenen Erkenntnisse und Ideen zur Fortsetzung dieser Arbeit formuliert.

1 Grundlagen

Nervensysteme haben die Aufgabe, Informationen über die Welt außerhalb eines Organismus zu sammeln und zu verarbeiten, um daraufhin Reaktionen auszulösen, die eine bestmögliche Anpassung an die Gegebenheiten der Umwelt des Lebewesens zur Folge haben. Die Neurowissenschaften sind ein Überbegriff für Wissenschaftsbereiche, die Nervensysteme zum gemeinsamen Studienobjekt haben. Biologie, Medizin, Physik und Psychologie tragen unter Verfolgung unterschiedlicher Ansätze und Methoden Erkenntnisse über ihre Funktionsweise und den ihnen zugrunde liegenden Aufbau bei.

Das interdisziplinäre *FACETS-Projekt*, im Rahmen dessen diese Arbeit angefertigt wurde, konzentriert sich dabei auf das Studium der in Nervensystemen stattfindenden Informationsverarbeitungprozesse. Aus den hieraus entstehenden Erkenntnissen erhofft man sich neue Rechenparadigmen, deren Anwendungen über die Möglichkeiten der herkömmlichen auf dem Turing-Modell basierenden IT-Systeme hinausreichen.

Dieses Kapitel soll eine kurze Einleitung in bestimmte Fragestellungen der Neurowissenschaften geben und den innerhalb des FACETS-Projekts verfolgten Ansatz vorstellen.

1.1 Neuromorphe Modellierung

Die Fähigkeit des Nervensystems, Information zu verarbeiten, ist auf seine Bestandteile, die *Neuronen* und ihre Verknüpfungen, die *Synapsen*, zurückzuführen. Ihre Funktionsweise bildet dabei die Grundlage der zugrundeliegenden Signalverarbeitung.

Neuronen bestehen aus einem Zellkörper oder *Soma* und den davon ausgehenden *Dendriten* und dem *Axon*. Über die Dendriten erhält ein Neuron Signale anderer Zellen, während es sein Axon benutzt, um selbst Signale zu übermitteln. Entlang der Zellmembran herrscht zu jeder Zeit ein Potenzialunterschied zwischen ihrer Innen- und Außenseite, der durch das Wirken von Ionenkanälen und -pumpen aufrecht erhalten wird. Überschreitet dieses *Membranpotenzial* einen Schwellwert, löst das Soma des Neurons eine kurzzeitige Veränderung des Membranpotenzials aus: das *Aktionspotenzial* (AP). Entlang des Axons wandert dieses *presynaptische* Aktionspotenzial in Richtung *Axonende*, bis es an die Synapsen gelangt, welche die Verbindung mit den Dendriten der *postsynaptischen* Neuronen herstellen. In den Synapsen wird die Ausschüttung sogenannter *Neurotransmitter* veranlasst, welche das Durchlassverhalten der in der postsynaptischen Zellmembran befindlichen Ionenkanäle steuert. Auf diese Weise verändert sich das Membranpotenzial des postsynaptischen Neurons, was letztendlich die Wahrscheinlichkeit für die Auslösung eines Aktionspotenzials in der nachgeschalteten Zelle beeinflusst. Was den qualitativen Effekt des presynaptischen AP auf der nachgeschalteten Zellmembran betrifft, gibt es *exzitatorische* und *inhibitorische* Synapsen, welche das Potenzial in Richtung des Schwellwerts verändern bzw. dieses von ihm entfernen.

Viele Neuronen bilden mittels synaptischer Verbindungen größere Gebilde, die *neuronalen Netze*. Die Eigenschaften eines solchen Netzes sind zum einen durch die oben erläuterte Funktionsweise der Neuronen gegeben, zum anderen entwickelt jedes Netzwerk eine eigene Dynamik, die nicht allein anhand seiner Bestandteile zu erklären ist, sondern aus der Struktur deren Vernetzung hervorgeht. Eine wichtige Aufgabe der Neurowissenschaften besteht nun darin, zu erkunden, welche Mechanismen dieser beobachteten Dynamik zugrunde liegen. Bei der Untersuchung des Verhaltens eines gegebenen Netzwerks stellt sich damit z.B. die Frage, welcher Anteil den lokalen Eigenschaften der Zellen und wieviel deren Vernetzung zugesprochen werden kann.

Der visuelle Kortex, ein evolutionär relativ junger Teil des Gehirns von Säugetieren, hat beispielsweise bei Ratten eine Neuronendichte von ca. 80.000 Neuronen pro mm^3 [45]. Jedes einzelne dieser Neurone besitzt durchschnittlich 7000 Synapsen [45], wodurch die Vernetzung der

Zellen ein komplexes Gebilde darstellt. Diese mikroskopischen Skalen machen es sehr schwierig, Informationen über z.B. den genauen Verlauf der vorhandenen Synapsen zu extrahieren. Man schätzt die Anzahl an Neuronen im menschlichen Kortex auf ca. 10^{10} und die der Synapsen auf 10^{14} [43], was die Aufgabe nochmals erschwert, die Funktionsweise und das Zusammenspiel der einzelnen Konstituenten zu verstehen.

Technisch ist es bis heute nicht zufriedenstellend realisierbar, ein neuronales Netz Synapse für Synapse zu untersuchen und somit seine Aktivitätsmuster auf allen Skalen nachzuvollziehen. Aus diesem Grund scheint die *Modellierung* einen vielversprechenden Ansatz darzustellen. Anstelle des "top down" Ansatzes geht man nach dem "bottom up"-Prinzip vor: man bildet die neuronalen Netze mit Hilfe von Modellen nach und versucht mit ihnen ein ähnliches Verhalten wie in der Natur zu erreichen. So kann man experimentell gewonnene Erkenntnisse nach und nach in die Modelle einbringen, um sukzessive die tatsächlich gemessenen Eigenschaften biologischer Netze zu reproduzieren. Dies erlaubt es beispielsweise der zuvor formulierten Frage nach der Beziehung zwischen der Rolle der einzelnen Zellen und der ihrer Vernetzung nachzugehen, da über diesen Ansatz die Auswirkungen einzelner modellierter Merkmale auf das Gesamtnetzwerk nachvollzogen und charakterisiert werden können.

1.1.1 Softwaresimulatoren und Neuromorphe Hardware

Neuronenmodelle bestehen üblicherweise aus einem Satz von Differentialgleichungen, die das Verhalten vor allem des Membranpotenzials und der synaptischen Prozesse quantitativ beschreiben. Ihre Lösungen können auf analytischem Wege gesucht werden, meistens ist jedoch die numerische Simulation praktikabler. Die dafür verwendeten *Softwaresimulatoren* (für einen Review-Artikel siehe [4]) bieten eine große Flexibilität, da der Komplexität der simulierten Modelle in Software grundsätzlich keine Grenzen gesetzt sind. Ferner besteht in numerischen Experimenten zu jedem Zeitpunkt Zugang zu allen Observablen.

Jedoch ist die Zeit, die numerische Simulatoren für die Berechnung benötigen, direkt abhängig vom Detail des zugrundeliegenden Neuronenmodells und der Konnektivität und Größe des Netzwerks.

Möchte man Simulationen großer Netzwerke, statistisch aussagekräftige Anzahlen von Simulationen oder Untersuchungen über einen längeren biologischen Zeitraum durchführen, so stößt man schnell an Grenzen bezüglich der Kosten bzw. des Energie- und Zeitaufwands (siehe z.B. [33]). Entscheidend ist dabei der Kontrast zwischen der Parallelität der in neuronalen Netzen stattfindenden Prozessen und der sequenziellen Natur von Computersimulationen, die auf einer relativ kleinen Anzahl an Prozessoren arbeiten.

Eine Alternative stellen die sogenannten *neuromorphen Hardwaresysteme* dar (siehe [16, 30]), die entscheidende Probleme der Softwaresimulatoren umgehen. Ziel ist es, das Verhalten von neuronalen Netzen zu imitieren. Anstatt mit digitaler Logik numerisch Differentialgleichungen zu lösen, werden die für Informationsverarbeitung relevanten Aspekte der Neuronen über die Wechselwirkung realer physikalischer Objekte nachgebildet. Um dies sprachlich von den Simulatoren zu unterscheiden, nennt man dieses Funktionsprinzip eine *Emulation*.

Der Vorteil von neuromorpher Hardware besteht hauptsächlich in seiner massiv parallelen, lokal analogen Funktionsweise. Unabhängig von Größe und Konnektivität des Netzwerkes emulieren neuromorphe Hardwaresysteme typischerweise in biologischer Echtzeit oder sogar um ein vielfaches beschleunigt [6].

Ein Nachteil stellt die eingeschränkte Flexibilität dar. Zum einen ist man fest an die in der Hardware verankerten Neuronen- und Synapsenmodelle gebunden und zum anderen lassen sich auch diese Parameter nur innerhalb eines während des Herstellungsprozesses festgelegten Bereiches variieren.

Dem steht der Vorteil der großen Geschwindigkeit der Experimente gegenüber. Neuromorphe Hardware ermöglicht die Skalierung der emulierten Netze unter Konstanthaltung der Berechnungszeit. Im Vergleich zur Leistung der Softwaresimulatoren stellt dies eine Beschleunigung dar, die mit steigender Netzwerkgröße an Bedeutung gewinnt. Aus diesem Grund eignen sich Hardwaremodelle beispielsweise dazu, für große Parameterbereiche schnelle Experimente durchzuführen und Daten für eine sinnvolle statistische Auswertung zu sammeln.

1.1.2 Die FACETS Stage 1 Hardware

Ein solches Neuromorphes Hardwaresystem wurde in der Arbeitsgruppe *Electronic Vision(s)* des Kirchhoff-Instituts im Rahmen des FACETS-Projekts erstellt: die FACETS Stage 1 Hardware, die im folgenden vorgestellt wird.

Die zuvor erwähnten physikalischen Objekte, die in einer Emulation das Verhalten von Neuronen und Synapsen imitieren, sind in diesem Fall die Elemente von elektronischen Schaltkreisen: Für die Emulation wird ein Mikrochip bzw. IC[7] verwendet. Auf diese Weise bestehen direkte Ähnlichkeiten in der Funktionsweise mit biologischen neuronalen Netzen, da in beiden Fällen elektrische Vorgänge auf sehr kleinem Raum stattfinden.

Das Kernstück der FACETS Stage 1 Hardware bildet ein Mikrochip der Bezeichnung *Spikey*, eine *Mixed-Signal-VLSI*-Architektur[8][38, 40]. Dieser beinhaltet insgesamt 384 Neuronen mit 100.000 programmierbaren Synapsen. In Planung steht die Möglichkeit, bis zu 16 Chips miteinander verbinden zu können um dadurch Netzwerke von mehreren Tausend Neuronen und Millionen Synapsen abbilden zu können. Momentan ist der Chip so konfiguriert, dass die Emulation bzgl. der biologischen Echtzeit einen Beschleunigungsfaktor von 10^4 aufweist, d.h. Vorgänge in Netzen, die in der Biologie 1 ms dauern, werden auf der Hardware in nur $0,1\,\mu$s emuliert. Diese hohe Beschleunigung ist durch die kleinen Zeitkonstanten der Analogschaltkreiselemente bedingt und damit eine intrinsische Eigenschaft des Netzwerkmodells auf der Hardware.

Abbildung 1 zeigt eine Mikrofotographie des $5 \times 5\,\mathrm{mm}^2$ einnehmenden *Spikey*-Chips. Der untere inhomogen strukturierte Teil besteht aus Digitallogik und verwaltet den Chip. In dem direkt darüber dunkel erkennbaren Spalt liegen die Neuronen, die in zwei Blöcke von je 192 eingeteilt sind. Die beiden grauen Rechtecke, welche den größten Teil der Chipfläche in Anspruch nehmen, sind die analogen *Synapsenarrays*. Die zu ihren Seiten befindlichen *Synapsentreiber* ordnen entweder einem externen Input, auch *virtuelles Neuron* genannt, oder einem auf dem rechten oder linken Block befindlichen Neuron eine Zeile des Synapsenarrays zu, von der aus die vertikale Verbindung zurück zu den postsynaptischen Neuronen hergestellt werden kann.

Der Spikey ist auf einem Trägerboard oder PCB[9] angebracht, auf dem außerdem ein FPGA[10] für Experiment- und Kommunikationskontrolle und ein RAM[11] zur Speicherung der notwendigen Experimentdaten sitzen. Ein Trägerboard besetzt einen von insgesamt 16 Steckplätzen auf einer Backplane, die über *Ethernet* mit einem Host-PC verbunden ist. Zusätzlich kann ein Oszilloskop an die Spikey-Chips angeschlossen werden, um die Membranspannung einzelner Neuronen direkt auszulesen (siehe Abbildung 2). Dadurch, dass der Spikey-Chip auf andere Einheiten wie den FPGA, das Trägerboard und die Backplane angewiesen ist, stößt man bei der Erweiterung der emulierten biologischen Netze auf eine weit größere Anzahl an Chips auf Kommunikationsprobleme. Allein der räumliche Abstand verursacht auf Grund der endlichen Signalausbreitungsgeschwindigkeit Verzögerungen in den Synapsen, die bei dem gegebenen Beschleunigungsfaktor

[7] Integrierter Schaltkreis
[8] Mixed-Signal bezieht sich auf die Verarbeitung analoger und digitaler Signale gleichzeitig, VLSI bedeutet *Very Large Scale Integration*.
[9] Printed Carrier Board
[10] Field Programmable Gate Array
[11] Random Access Memory

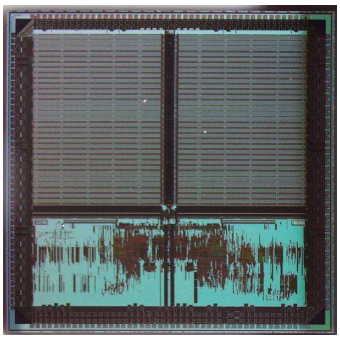

Abbildung 1: Mikroaufnahme des *Spikey*-Chips: Über dem unteren Digitalteil befinden sich in zwei Blöcke eingeteilt 384 Neuronen, deren 100.000 mögliche Synapsen über die zwei analogen *Synapsenarrays* im oberen Teil des Chips realisiert werden.

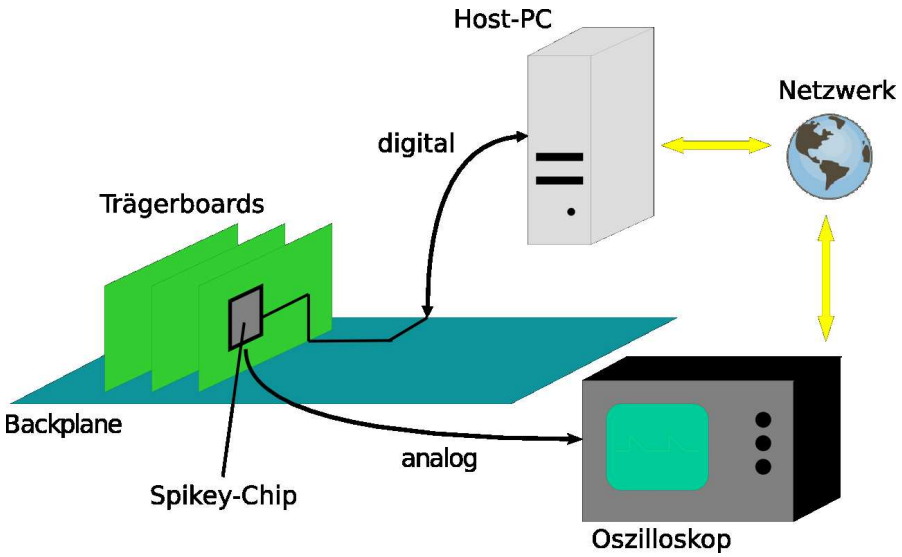

Abbildung 2: Das FACETS Stage 1 Hardware System: Der Spikey-Chip sitzt zusammen mit einem FPGA und dem RAM auf einem Trägerboard, von denen mehrere über eine Backplane digital mit dem Host-PC kommunizieren. Zusätzlich kann ein mit einem Spikey analog verbundenes Oszilloskop einzelne Membranpotenziale auslesen. Der Host-PC wird über das Netzwerk von einem beliebigen Standort aus gesteuert. Abbildung mit freundlicher Erlaubnis von Daniel Brüderle.

nicht zu kompensieren sind. Im Bestreben, eine noch größere Anzahl an Neuronen und Synapsen zu simulieren, wird die sogenannte FACETS *Stage 2* Hardware entwickelt [39]. Diese wird mit Hilfe der neuartigen *Wafer-Scale-Integration* hergestellt. Die programmierbaren analogen neuronalen Netze können auf diese Weise nebeneinander auf dem Siliziumwafer positioniert und miteinander verbunden werden. Ca. 350 HICANNs (High Input Count Analog Neural Network) mit jeweils 512 Neuronen und 512×256 Synapsen finden auf einem Siliziumwafer Platz, sodass idealerweise mehr als 180.000 Neuronen und 45 Millionen Synapsen auf einem Wafer emuliert werden könnten. Auf Grund der hohen Ähnlichkeit der HICANNs zum Spikey-Chip können viele der mit der Vorgängerversion gesammelten Erfahrungen in die Entwicklung des neuen Systems eingebracht werden.

Das Hardware Neuronenmodell

Das von der FACETS Stage 1 Hardware implementierte Neuronenmodell baut auf dem konduktanzbasierten *Leaky Integrate-and-Fire-Neuron* [10] auf, das über eine die Dynamik seines Membranpotenzials beschreibende Differentialgleichung definiert ist. Diese wird im folgenden kurz erwähnt mit Verweis auf [13] und [10], wo eine genaue Erklärung der Bedeutung der jeweiligen Parameter zu finden ist. Das Membranpotenzial V_m der auf *Spikey* implementierten Neuronen verhält sich gemäß:

$$-C_\mathrm{m}\frac{\partial V_\mathrm{m}}{\partial t} = g_\mathrm{leak}(V_\mathrm{m} - V_\mathrm{rest}) + \sum_e g_e(t)(V_\mathrm{m} - V_\mathrm{rev,exc}) + \sum_i g_i(V_\mathrm{m} - V_\mathrm{rev,inh}) \tag{1}$$

wobei $V_\mathrm{rest,exc}$, $V_\mathrm{rest,inh}$ und V_rest für das exzitatorische, inhibitorische Ruhe- bzw. Umkehrpotenzial stehen. Die Summen laufen über alle exzitatorischen (e) und inhibitorischen (i) Synapsen, deren Konduktanzen die synaptischen Ionenströme steuern. Wird dem Synapsentreiber ein Aktionspotenzial für das entsprechende Neuron digital gemeldet, bestimmt eine Kette von Mechanismen die Veränderung der synaptischen Konduktanz g_e bzw. g_i. Zu Beginn steigt g sprungartig auf einen Anfangswert, um dann exponentiell mit der Zeitkonstante τ_rise bis zu einem Wert $w \cdot g_\mathrm{max}$ weiter anzuwachsen. w ist dabei das sogenannte *synaptische Gewicht* und g_max einstellbar für je ein presynaptisches Neuron. Anschließend klingt g exponentiell mit τ_fall wieder ab.

Des weiteren besitzt jedes Neuron eine Membrankapazität C_m und einen Ladungsstrom hin zum Ruhepotenzial gemäß der Konduktanz g_leak. Überschreitet V_m die Schwellspannung V_thresh (siehe Einleitung von Abschnitt 1.1), wird ein digitales Aktionspotenzial (ein sogenannter *Spike*) ausgelöst und gleichzeitig das Membranpotenzial auf den Wert V_reset gesetzt, wo es für eine Refraktärzeit τ_refrac verweilt, bis es wieder der Membrandynamik freigegeben wird und ohne synaptische Ionenströme auf den Wert V_rest zurückkehrt. auf den Wert V_rest zurückkehrt. Auf der Hardware ist die Integrationsfähigkeit der Neuronen in Form von Kondensatoren realisiert, die von einem Vergleichsschaltkreis hinsichtlich Schwellüberschreitung überwacht werden. Geschieht dies, wird das Ereignis dem Digitalteil gemeldet, der die Refraktärzeit berücksichtigt und anschließend den entstandenen Spike registriert und gegebenenfalls weiterleitet.

1.1.3 Die Metasprache PyNN

Um Experimente auf dem FACETS Stage 1 System durchführen zu können, gibt es einen Softwarestapel, der zwischen der Hardware auf niedrigster und dem Benutzer auf höchster Ebene vermittelt.

Die unmittelbare Kommunikation mit der Hardware wird von C- und C++-Programmen gesteuert [17, 20]. Diese werden von sogenannten *Wrapperklassen* [1] für die interpretierte Programmiersprache *Python* [37] innerhalb des Moduls PyHAL[12] zugänglich gemacht. An oberster

[12] Python Hardware Abstraction Layer

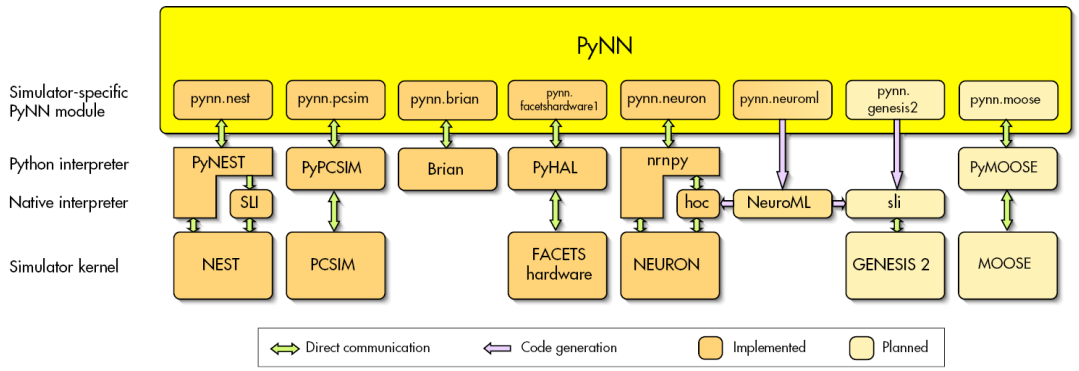

Abbildung 3: Schematische Darstellung der Struktur von PyNN mit den unterstützten Simulatoren. Abbildung mit freundlicher Erlaubnis von Daniel Brüderle.

Stelle dieser hierarchisch angeordneten Softwareebenen steht das *Hardware PyNN-Modul* [9], in dem der Benutzer die durchzuführenden Experimente definiert.

Die Metasprache PyNN[13] (ausgesprochen wie *pine* im Englischen) [8] ist ein Python-Modul, das eine simulatorunabhängige Spezifikation von neuronalen Experimenten erlaubt. Die Beschreibung von dem zu simulierenden biologischen Netzwerkmodell wird der Benutzerschnittstelle von PyNN einmalig übergeben, um dann von simulatorspezifischen PyNN-Untermodulen in die jeweiligen sich unterscheidenden Schnittstellenspezifikationen übersetzt zu werden. Unterstützt wird eine große Anzahl an Softwaresimulatoren, die in Abbildung 3 zu sehen sind. Im Fall der FACETS Stage 1 Hardware ist es das Modul `pyNN.facets.hardware1`, das zwischen dem übergreifenden PyNN-Modul und der PyHAL vermittelt.

Die PyNN-API[14] bietet dem Benutzer zwei sich ergänzende Arten der Netzwerkbeschreibung an. Einmal ist es die Low-Level-API, mit der prozedural die Eigenschaften des zu simulierenden Netzwerkes übergeben werden (zur Verfügung stehen beispielsweise die Funktionen `create()`, `connect()` und `set()`). Sie eignet sich für eine geringe Anzahl von Neuronen und Synapsen, bei denen die manuelle Einstellung der einzelnen Parameter gewünscht ist. Wenn jedoch komplexere Netzwerke simuliert werden und die Details im Hintergrund stehen, bietet sich die objektorientierte High-Level-API an. Die Klasse `Population` erlaubt es, ähnliche Neuronen zu gruppieren, um diese dann mit der Klasse `Projection` auf verschiedenste Weise zu verbinden. Je nach Bedarf können auch beide Schnittstellen gemeinsam verwendet werden.

Der Hauptvorteil von PyNN ergibt sich durch die Portabilität der Experimente. Ergebnisse sind somit transparent und können ohne zusätzlichen Aufwand auf anderen Simulatoren überprüft werden. Im Falle der Hardware ist die Einbindung in PyNN auf Grund ihrer völlig unterschiedlichen Funktionsweise und dem daraus folgenden Bedarf am Vergleich mit den bereits etablierten Simulatoren von besonderer Bedeutung. So ist z.B. die Differentialglg. 1 als Neuronenmodell im PyNN-Untermodul des Softwaresimulators NEST[15] [11] implementiert und somit kann dieser als Referenz für Experimente auf der Hardware verwendet werden.

[13] A Python package for simulator-independent specification of neuronal network models
[14] Application Programming Interface
[15] Neural Simulation Technology

1.1.4 Mapping Software und Graphenmodell

Eine der wichtigsten und anspruchsvollsten Aufgaben der in PyHAL enthaltenen Software besteht darin, das vom Benutzer bzw. von PyNN vorgegebene biologische Netzwerkmodell (im folgenden BN) in eine entsprechende Konfiguration der Hardware umzuwandeln. Dabei muss primär die Struktur des BN in die fest vorgegebene Topologie und Parameterbereiche des Hardwaresystems übersetzt werden. Die Neuronen und Synapsen müssen so gesetzt werden, dass das sich auf Hardware ergebende Netzwerk (HN) möglichst gut das zugrundeliegende BN wiedergibt, wobei gleichzeitig Optimierungsaspekte beachtet werden müssen. Zu diesem Zweck wurde für die PyHAL des FACETS Stage 2 Hardware eine komplexe *Mapping Software* erstellt. In Hinsicht auf die geplante gleichzeitige Verwendung mehrerer Spikey-Chips wird dieses Übersetzungssystem momentan auch für die FACETS Stage 1 Hardware implementiert.

Für die Repräsentation von Daten im Rahmen der Mapping Software wurde ein eigenes *Graphenmodell* (GM) [47] in objektorientiertem C++ entwickelt. Dem BN und dem HN werden entsprechende Darstellungen im GM zugeordnet, der *Biograph* (BG) und der *Hardwaregraph* (HG). Diese Darstellungen bestehen aus sogenannten *Knoten* und *Kanten* (siehe Abbildung 4), die jeweils als Klassen implementiert sind. Folgende unterschiedliche Kanten gibt es:

- *hierarchische* Kanten legen die Hierarchie der Knoten fest, an deren Spitze der sogenannte SystemNode steht
- *gerichtete* Kanten verknüpfen zwei beliebige Knoten miteinander und können durch unterschiedliche Namen die Bedeutung ihrer Verbindung charakterisieren
- *Hyperkanten* gehen von gerichteten Kanten aus und enden auf Knoten.

Die innerhalb des Graphenmodells definierten Klassen enthalten eine Vielzahl von Methoden, um Manipulationen an einem Graphen durchzuführen. Zusätzlich wurde eine eigene Pfadsprache entwickelt [46], um sich innerhalb des Graphen fortbewegen zu können. Für diese Funktionalitäten wurden im Rahmen dieser Arbeit Komponententests entwickelt (siehe Abschnitt 2.1.1).

Die Aufgabe der Mapping Software ist nun klarer definiert: Zu einem gegebenen BG muss ein HG erstellt werden, der zum einen dessen Aufbau möglichst treu wiedergibt und zweitens eine Hardwarekonfiguration darstellt, welche die auf der Hardware vorhandenen Verbindungsbandbreiten und Parameterbereiche möglichst optimal zum Einsatz bringt. Dafür wird ein rekursiver Algorithmus verwendet, der die hierarchische Struktur des Biographen schrittweise durchläuft und dementsprechend den Hardwaregraphen schrittweise vervollständigt. Die Minimierung sogenannter *Kostenfunktionen* ist ausschlaggebend, für welche der möglichen Ausprägungen des Hardwaregraphen sich die Software jeweils entscheidet [47].

1.2 Software-Tests

Unit tests, oder auch Komponententests genannt, dienen dazu, die korrekte Funktionalität von einem Teil des Gesamtcodes eines Softwareprojekts sicherzustellen. Dieser Quellcode kann idealerweise in einzelne Module eingeteilt werden, die über eine Schnittstelle miteinander kommunizieren.

1.2.1 Modularisierung

Mit der Einführung der objektorientierten Progammierung, die im Kontrast zur prozeduralen Programmierung steht, ist diese Technik der Einteilung in Unterkomponenten (siehe Abbildung 5) zur Praxis geworden [31]. Die Vorteile eines stark modularisierten Programmierstils liegen in

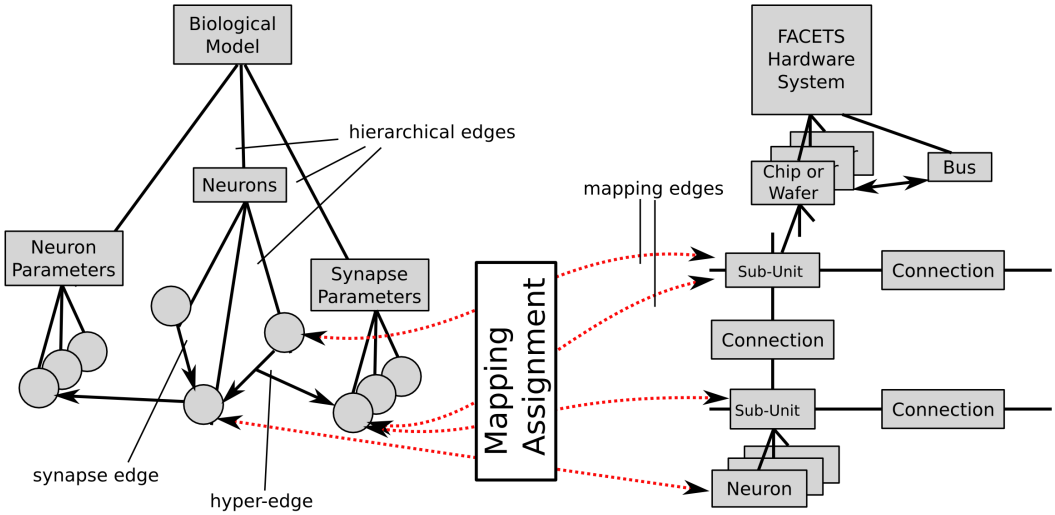

Abbildung 4: Das Graphenmodell dient der Informationsdarstellung im Rahmen des Mappingprozesses. Abbildung mit freundlicher Erlaubnis von Daniel Brüderle.

der konsequenten Strukturierung, die dieser ermöglicht bzw. erzwingt. Der Aufbau des Gesamtprogramms wird im Vorfeld des Entstehungsprozesses definiert, d.h. die Spezifikationen der nötigen Module und Schnittstellen stehen von Beginn an fest. Die Implementierung des Codes kann auf diese Weise unter den Entwicklern besser aufgeteilt und deshalb unabhängig voneinander durchgeführt werden. Auch bei der Wartung des Codes, die oft den größten Teil des Arbeitsaufwands darstellt, ist die Modularisierung von Vorteil. Dadurch, dass nur die Schnittstelle eines Moduls den Kontakt zu anderen Teilen des Programms definiert, können die einzelnen Komponenten unabhängig voneinander getestet und korrigiert werden.

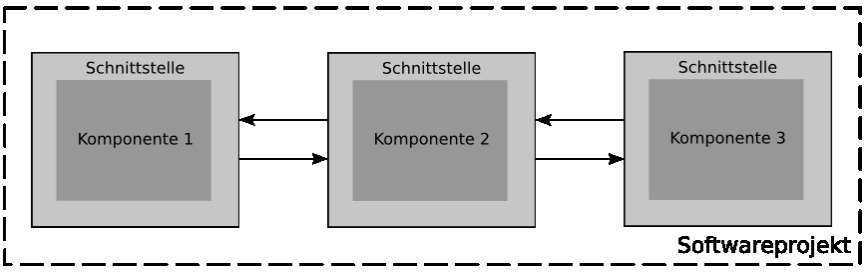

Abbildung 5: Modularisiertes Softwareprojekt

1.2.2 Komponententests

Der Test einer Komponente bedeutet im weitesten Sinne das Überprüfen ihrer korrekten Funktionalität. Korrekt bedeutet in diesem Kontext, dass das Verhalten des Moduls exakt seinen vorgegebenen Spezifikationen entspricht. Tut man dies systematisch, stellen Komponententests (oder auch Unit Tests genannt) einen möglichen Ansatz hierfür dar. Sie sollten die Eigenschaft

Abbildung 6: Testen nach dem Blackboxprinzip

haben, isoliert und zu beliebigen Zeitpunkten ausführbar zu sein. Für die Isolierung ist es wichtig, die Tests nicht invasiv durchzuführen. Wird der testende in den zu testenden Code eingepflegt, besteht das Risiko, durch diesen Eingriff seinerseits unerkannt Fehler zu erzeugen. Sinnvoll ist es, die Implementierung der Tests auf der gleichen Ebene anzusiedeln, in der auch benachbarte Module auf die Schnittstelle zugreifen [15].

Dies legt dabei nahe, sich nicht um die genaue Implementierung des zu testenden Moduls zu kümmern, da die mit ihm in Verbindung stehenden Komponenten einzig über die Schnittstelle auf seine Funktionalität zugreifen und man somit erreicht, dass Testszenarien möglichst treu tatsächliches Verhalten des Programms simulieren (siehe Abbildung 6) Dies entspricht dem sogenannten Blackboxprinzip [44]. Man präsentiert dem zu testenden Programm hierbei eine vordefinierte Eingabe, sogenannte Testfälle, und erkennt anhand der Ausgabe, ob das gewünschte Verhalten eintritt. In den seltensten Fällen ist es möglich, die Gesamtheit der möglichen Eingaben zu überprüfen. Wird dem Programm beispielsweise eine Zeichenkette der Länge n übergeben, sind bei einer Zeichengröße von 8 Bit bereits 256^n Eingaben möglich. Da man auf diese Weise schnell an zeitliche Grenzen stößt, muss man sich meist exemplarisch bzw. stichpunktartig auf bestimmte Testfälle beschränken. Da aber die genaue Implementierung des Moduls nicht bekannt ist, kann nicht sichergestellt werden, dass jeder Teil des Codes durchlaufen wird. Stattdessen richtet man sich nach der Spezifikation und wählt die Testfälle so, dass das vorgegebene Verhalten verifiziert wird.

Automatisiert man zusätzlich dazu die Durchführung der Tests, so bestätigen sie nicht nur die Richtigkeit des Codes zum gegenwärtigen Zeitpunkt, sondern können dies auch in Zukunft sicherstellen. Dadurch, dass die Testfälle allein auf Grund der vorhandenen Information über die Schnittstelle definiert sind, müssen die Tests zu jedem Zeitpunkt bestanden werden. Auf diese Weise kann beispielsweise die Behebung bestimmter Fehler dokumentiert werden, indem Testfälle die ursprünglich fehlerproduzierende Eingabe enthalten und somit erst nach erfolgreicher Korrektur sinnvolle Testausgänge bewirken. Außerdem wird durch die ständige Ausführung der Tests während der weiteren Entwicklung des Codes erzwungen, dass die bereits getesteten Designvorgaben genau eingehalten werden. In der sogenannten testgesteuerten Entwicklung macht man sich diese Eigenschaft zunutze und schreibt die Tests noch vor dem Code selbst. Auf diese Weise erfüllen die Tests zusätzlich zu ihrer qualitätssichernden Funktion die Aufgaben der Designvorgabe und der Dokumentation des Codes.

1.2.3 Testumgebung CppUnit

Eine Umgebung, die Komponententests mit diesen Eigenschaften möglich macht, ist `CppUnit`[16]. Für die C++ Komponenten der FACETS-Software (siehe Abschnitt 1.1.4) wurde dieses Tool ein-

[16] `http://sourceforge.net/projects/cppunit/`

heitlich gewählt, da es als eines der weitverbreitetsten Frameworks für C++ eine unkomplizierte Handhabung erlaubt. In `CppUnit` wird eine Klasse erstellt, die von einer von `CppUnit` bereitgestellten Klasse `TestFixture` erbt und mit `CppUnit`-Makros erweitert wird. In dieser Klasse können nun Methoden definiert werden, welche die einzelnen Tests enthalten. Der Test selbst, also die Vergleiche auf unterster Ebene aus Sicht des Anwenders sind `CppUnit`-Makros, die letztendlich die Funktionalität zur Überprüfung von Testannahmen bereitstellen. Außerdem stellt die Klasse `TestFixture` zwei Methoden zur Verfügung, `setUp` und `tearDown`. `setUp` wird vor jedem Test aufgerufen und erlaubt damit, zu Beginn Instanzen der Klasse anzulegen, an der die Tests durchgeführt werden sollen. Nach jedem Test stellt die Methode `tearDown` sicher, dass der zuvor angelegte Speicherplatz wieder freigegeben wird. Auf diese Weise können die Tests unabhängig voneinander mit denselben Anfangsbedingungen durchgeführt werden. Mit Hilfe der Makros werden sie letztendlich registriert und zu sogenannten `test suites` hinzugefügt, welche dann in einer Test-`main`-Funktion aufgerufen werden.

1.3 Liquid Computing

Der Abschnitt 1.1 handelt von der Modellierung der einzelnen Neuronen und Synapsen und die Umsetzung davon auf der FACETS Stage 1 Hardware. Für die tatsächliche Verarbeitung von Information wird jedoch ein Modell benötigt, welches dem Netzwerk als Ganzes einen Rahmen gibt, innerhalb dessen es Berechnungen durchführen und seine Fähigkeit zur Informationsverarbeitung evaluiert werden kann. Im folgenden wird ein solches Modell vorgestellt, das ähnlich wie das Turingmodell einen Berechnungsvorgang definiert, dafür jedoch auf einem grundlegend unterschiedlichen Ansatz basiert: Das sogenannte *Liquid Computing*, wie von W. Maass et al. in [24] und H. Jaeger in [18] unabhängig voneinander präsentiert.

Liquid Computing als unkonventionelles Modell

Die Entstehung von Liquid Computing ist auf eine biologische Motivation zurückzuführen. Es ist bekannt, dass der Neokortex von Säugetieren konstant von zeitlich stark variierenden sensorischen Informationen über die Umwelt gespeist wird [24]. Obwohl die neuronalen Netze des Neokortex gewisse Gemeinsamkeiten besitzen, handelt es sich um stark rückgekoppelte und dadurch hochkomplexe Netzwerke [12]. Erstaunlich ist dabei die Vielfalt und Komplexität der Verarbeitung des erhaltenen Inputs in Echtzeit. Herkömmliche Ansätze zur Modellierung der Informationsverarbeitung in neuronalen Netzen, beispielsweise Attraktormodelle [7] oder das Turingmodell, setzen die Kenntnis des vorliegenden Netzes voraus und stoßen deshalb bei der Analyse komplexer Netzwerke an Grenzen. Liquid Computing dagegen macht sich diese Komplexität zunutze, indem ein neuronales Netz den erhaltenen Input generisch verarbeitet und erst im zweiten Schritt das Ergebnis davon aufgabenspezifisch ausgewertet wird.

Des weiteren verarbeitet Liquid Computing den Input in Echtzeit und stellt den daraus resultierenden Output zu jedem Zeitpunkt zur Verfügung. Dabei steht seine parallele Funktionsweise im Gegensatz zur sequenziellen Verarbeitung herkömmlicher, auf dem Turingmodell basierender Computer.

Die Funktionsweise einer Liquid State Machine

Anschaulich ist eine LSM, wie von Maass et al. in [24] dargelegt, wie folgt aufgebaut: Ein Medium wird durch einen Input angeregt, woraufhin es verschiedene Zustände annimmt. Die Konstituenten des Mediums wechselwirken miteinander und bewirken damit, dass zu einem späteren Zeitpunkt noch Spuren des zuvor angelegenen Inputs vorliegen. Ein Beobachter wird auf die Erkennung dieser Spuren trainiert und kann so zu einem gegebenen Zeitpunkt Rückschlüsse auf Inputs aus der Vergangenheit ziehen.

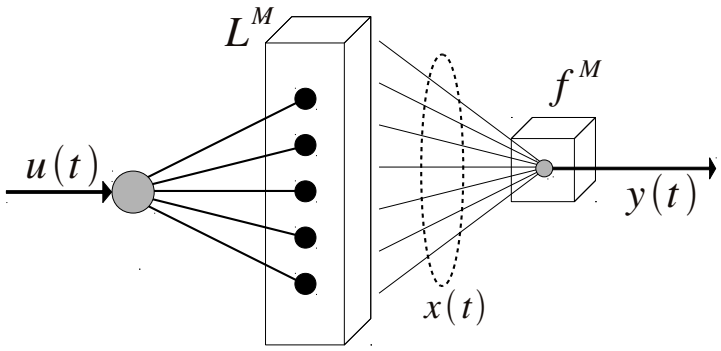

Abbildung 7: Schema der Struktur einer LSM: Der Input $u(t)$ wird in das Liquid L^{M} eingekoppelt, wo er verarbeitet wird. Der Zustand $x(t)$ des Liquids wird an den Klassifizierer f^{M} übergeben, der den Output $y(t)$ berechnet.

Formeller betrachtet wird ein von der Zeit abhängiger Input $u(t)$ kontinuierlich in eine Verarbeitungseinheit, das *Liquid*, eingespeist. Dort wird dieser von einem Filter L^{M} verarbeitet, sodass sich ein daraus resultierender Zustand $x(t)$ ergibt. Eine Klassifizierungsfunktion f^{M} wandelt diesen in den Output $y(t)$ um, wobei sie nur vom aktuell vorliegenden Zustand abhängt:

$$y(t) = f^{\mathrm{M}}(L^{\mathrm{M}}(u(\cdot)))\,. \tag{2}$$

Die Klassifizierungsfunktion f^M kann mit Hilfe eines Lernalgorithmus auf unterschiedliche Aufgaben bzw. Berechnungen trainiert werden. Um sie zu trainieren wird iterativ ein bekannter Input angelegt. Für jeden Iterationsschritt wird die Wirkung von f^M auf den Zustand $x(t)$ modifiziert. Der resultierende Output nähert sich dabei schrittweise dem für die präsentierten Trainingseinheiten gewünschten Wert (für ein Review über Lernalgorithmen siehe [21]). Da der Klassifizierer den Zustand des Liquids nur ausliest, ihn dabei aber nicht beeinflusst, ist er von diesem unabhängig. Dadurch können mit einem Liquid beliebig viele Klassifizierer verbunden sein, die auf jeweils unterschiedliche Aufgaben trainiert sind.

Maass et al. haben im Rahmen von theoretischen Betrachtungen gezeigt, dass auch im Fall eines linearen f^{M} die LSM jede Funktion annähern kann, die auch von einem endlichen Zustandsautomaten berechnet wird. Voraussetzung dafür ist ein genügend komplexes Liquid, das den Input in einen ausreichend hochdimensionalen Zustandsraum projiziert. Die sogenannte *Separationseigenschaft* gibt dabei an, wie stark unterschiedliche Inputs auch unterschiedliche Zustände des Liquids zur Folge haben. Für eine Quantifizierung dessen definiert man jeweils für die Inputs und die Liquidzustände eine Abstandsfunktion. Das Verhältnis zwischen den Abständen im Inputraum und denen im Zustandsraum ist nach [24] ein Indikator für die Klassifizierungsleistung der LSM. Des weiteren beschreibt die sogenannte *Approximationseigenschaft* die Fähigkeit des Klassifizierers, verschiedene Zustände des Liquids aufzulösen bzw. in deren zugehörigen Output umzuwandeln. Sie hängt von der Anpassbarkeit des Klassifizierers an die verschiedenen Aufgaben ab [24]. Da der Klassifizierer kein Erinnerungsvermögen besitzt, ist die Verarbeitung von in der Vergangenheit liegenden Inputs eine weitere Eigenschaft der LSM: Man spricht vom *schwindenden Speicher* des Liquids.

Die LSM als neuronales Netz

Das oben beschriebene Modell trifft auf eine Vielzahl von denkbaren Realisierungen einer LSM

zu. So wird z.B. in [35] eine physikalische Flüssigkeit, die der Input in Form von mechanischen Wellen anregt, als Liquid verwendet. Ein Klassifizierer in Software wertet dabei zu jedem Zeitpunkt ein Kamerabild der Flüssigkeitsoberfläche aus, das somit als Zustand der LSM dient. Natschläger et al. zeigen, dass die Realisierung dieser nur aus Anschauungszwecken gewählten Konfiguration bereits eine funktionierende LSM darstellt.

Ein weiteres Medium, auf dem Liquid Computing durchgeführt werden kann, sind neuronale Netze. Auf Grund ihrer Fähigkeit, komplexe Reizmuster auszubilden, eignen sie sich sehr gut als Träger des hochdimensionalen Liquidzustandes. Ein Netzwerk von Integrate-and-Fire Neuronen wurde von den Autoren von [24] als LSM umgesetzt und in Software simuliert, wobei die Berechnungs- und Speicherfähigkeit dieses Systems demonstriert wurde.

Trotzdem ist die Eignung von neuronalen Netzen zur Realisierung einer Liquid State Machine laut der Analyse von Bertschinger et al. in [3] nicht unmittelbar gegeben. Vielmehr ist sie an gewisse Bedingungen gebunden. Ein zweidimensional von Inputeigenschaften und Konnektivitätsparametern aufgespannter Raum teilt sich nach Bertschinger et al. in zwei Bereiche auf, denen zwei unterschiedliche Verhaltensweisen des Netzwerks entsprechen. Im ersten Bereich ist überwiegend deterministisches bzw. direkt vom Input bestimmtes Verhalten zu beobachten. Das Liquid besitzt in diesem Fall eine sehr kleine Speicherkapazität und spiegelt fast ausschließlich den momentan anliegenden Input wieder. Befindet es sich im zweiten Bereich, tritt das Gegenteil ein und ein sehr dynamisches Verhalten kommt zum Vorschein, das weitgehend unabhängig vom Input ist. In beiden Fällen enthält der Zustand $x(t)$ nur sehr wenig Information. Um die Leistung des Liquids hinsichtlich Speicherkapazität und Berechnungsfähigkeit zu optimieren, muss sich das Liquid im Übergangsbereich zwischen diesen beiden charakteristischen Dynamikregimes befinden [3]. In der englischsprachigen Literatur ist dieser Übergang nach C.G. Langton [22] als *„edge of chaos“* bekannt. Eine Erkundung dieser Typen von Dynamiken wurde in [42] unter Verwendung von binär arbeitenden McCulloch-Pitts-Neuronen [29] auf einem Mixed-Signal-IC vorgenommen.

2 Inbetriebnahme

2.1 Unit Tests

Im folgenden werden die Unit Tests vorgestellt, die im Rahmen dieser Arbeit erstellt wurden.

2.1.1 Tests am Graphenmodell

Wie in den Grundlagen in Abschnitt 1.1.4 beschrieben wird das Graphenmodell verwendet, um während des Mappingprozesses zu verarbeitende Informationen zu repräsentieren. Ausgehend von einem "Graphen", der das neuronale Netz in seinen biologischen Eigenschaften beschreibt, wird ein zweiter Graph, der die Hardware mit all ihren Konfigurationsmöglichkeiten enthält, manipuliert, bis er das biologische Netzwerk möglichst präzise wiedergibt. Die dabei benutzten Datenstrukturen, also Knoten, Kanten und der Graph als Ganzes werden in der objektorientierten Implementierung in C++ jeweils als Klassen dargestellt. Durch die Methoden dieser Klassen kann das Graphenmodell manipuliert werden. Knoten und Kanten haben ohne einen sie enthaltenden Graphen keine Bedeutung, so dass sich ihre zugehörigen Klassendefinitionen innerhalb der Klasse `GraphModel` befinden: Es gibt also `GraphModel`, `GraphModel::GMNode` und `GraphModel::GMConnection` [47]. Hyperkanten besitzen keine eigene Klasse, sondern erscheinen in der Beschreibung der Kante, von der sie ausgehen.

Abbildung 8 zeigt einen Beispielgraphen, an dem die Mehrheit der im Rahmen der hier präsentierten Arbeit entwickelten Tests durchgeführt werden. Er ist so gewählt, dass er zum einen zwecks Klarheit übersichtlich ist, zum anderen jedoch verschiedenste Anordnungen der Datenstrukturen enthält, um etwaige Grenzfälle abzudecken bzw. eine repräsentative Auswahl der tatsächlich vorkommenden Verzweigungen darzustellen. Um Auswirkungen möglicher Fehler in der Erstellung dieses Testgraphen zu vermeiden, werden nicht die üblichen Methoden zum Anlegen neuer Knoten und Kanten verwendet. Stattdessen wird der Graph manuell konstruiert, d.h. nur die Standardkonstruktoren kommen zum Einsatz. Die Erstellung geschieht in der oben vorgestellten `setUp`-Funktion, während dementsprechend die `tearDown`-Funktion den angelegten Speicherplatz nach jedem Test an dem Beispielgraphen wieder freigibt.

Der folgende Codeausschnitt soll beispielhaft den Test der Methode `EraseConnection()` illustrieren:

```
 1:void GraphModel_test :: testEraseConnection ()
 2:{
 3:
 4:  // erase N1_121, the named edge from K1 to K121 with attribute to K11:
 5:  graph_model->EraseConnection ( *N1_121 );
 6:
 7:  // check connection vectors:
 8:  CPPUNIT_ASSERT_EQUAL ( (int) 0 , (int) K121->InConnections->size() );
 9:  CPPUNIT_ASSERT_EQUAL ( (int) 1 , (int) K1->OutConnections->size() );
10:  CPPUNIT_ASSERT_EQUAL ( (int) 1 , (int) K11->AttributeConnections->size() );
11:  N1_121 = new GraphModel::GMConnection;
12:
13:}
```

Im Beispielgraph aus Abbildung 8, an dem die Tests durchgeführt werden, besitzt der Knoten `K1` eine Kante des Typs `ABSTRACT_HWCOMM`, die auf `K121` zeigt und gleichzeitig als Hyperkante auf `K11` gerichtet ist. Die Knoten `K1`, `K11` und `K121` besitzen Vektoren, in denen die eingehenden und ausgehenden Kanten bzw. Hyperkanten aufgelistet sind. Nachdem die Kante in Zeile 5 mit Hilfe der zu testenden Funktion gelöscht worden ist, sollte sie auch in den entsprechenden

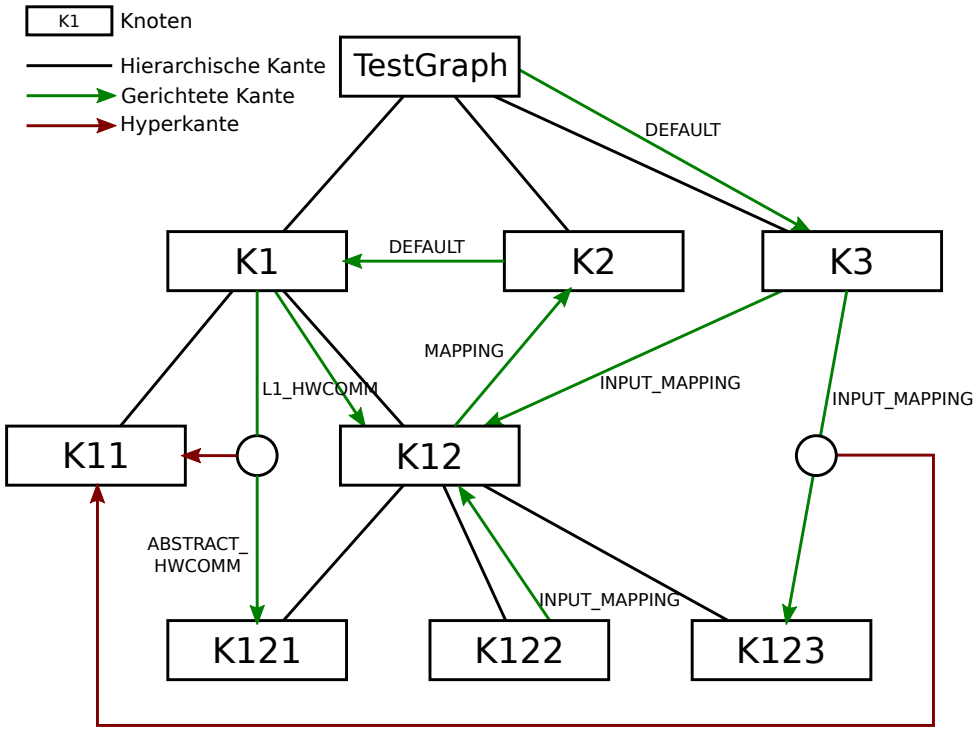

Abbildung 8: Der Testgraph: Seine Knoten und Kanten werden vor jeder Testfunktion angelegt und im Nachhinein wieder gelöscht. An ihm kann die korrekte Funktionsweise der zur Verfügung stellenden Funktionen überprüft werden.

Vektoren nicht mehr vorkommen und damit die Länge dieser angepasst werden. Die *CppUnit-Assertions* in den Zeilen 8 bis 10 überprüfen dies: `K121` sollte keine eingehende Kante mehr besitzen, von `K1` ginge nur noch die Kante nach `K12` aus und auf `K11` sollte keine Hyperkante mehr gerichtet sein. Zum Schluss wird das zuvor zerstörte Kantenobjekt wieder angelegt, da die `tearDown()`-Funktion den kompletten Beispielgraphen nach jedem Test zerstört und ansonsten einen ungültigen Speicherzugriff vornehmen würde.

2.1.2 Ergebnisse und Testkonzepte

Die Durchführung der Tests verifizierte den Großteil des aktuellen Codes. Dies war auf Grund der Tatsache, dass der Mappingprozess bereits funktionstüchtige Hardwaregraphen erzeugte, prinzipiell zu erwarten. Für vereinzelte Methoden konnte ein Fehlverhalten aufgedeckt werden. So war es z.B. innerhalb der Pfadsprache nicht möglich, den sogenannten *System Node* als Startpunkt zu verwenden. Dieser und weitere Fehler wurden in das Projektverwaltungssystem Indefero eingetragen (siehe Anhang A.1). Dort werden in Form von *Issues* ausstehende Aufgaben und zu behebende Fehler eingetragen und *Milestones* zugeordnet. Grundsätzlich sind die Tests unabhängig von der genauen Implementierung des Graphenmodells. Somit können die Tests ihre wichtigste Aufgabe erfüllen: Jede zukünftige Änderungen am Code des Graphenmodells, welche die gegenwärtig zur Verfügung stehende Funktionalität betrifft, kann jederzeit verifiziert werden.

Neben der Erstellung und Implementierung der Tests muss darauf geachtet werden, dass ihre vorgesehenen Aufgaben tatsächlich in der Praxis erfüllt werden. Die ständige Verifizierung

geschieht nur, wenn die Tests in einen passenden Rahmen eingebunden werden, der dies ermöglicht.

Die Umsetzung eines sinnvollen *Testflows* sollte damit beginnen, jeden beteiligten Entwickler mit der Erstellung der Tests und der gewählten Testumgebung vertraut zu machen. Des weiteren kann eine Automatisierung der Durchführung der Tests angestrebt werden. So kann beispielsweise bei Benutzung eines Programms zur Versionskontrolle vor jedem *commit* überprüft werden, ob die jeweilige Komponente nach der vorgenommenen Änderung ihre Tests noch besteht. Außerdem sollte bei der Erstellung und Integration neuer Tests auf Einfachheit geachtet werden. Daher stellte sich das Tool CppUnit im Rahmen der in dieser Arbeit beschriebenen Unit Tests als sehr nützlich heraus und könnte als die Standardtestumgebung für C++-Code verwendet werden. Des weiteren ist ein *Framework* in Entwicklung, das die Übersicht über die bereits vorhandenen Tests vereinfacht und deren Aufrufe vereinheitlicht.

2.2 Systemtests und Kalibrierung der Hardware

In Kapitel 3 werden die Ergebnisse von Experimenten auf der FACETS Stage 1 Hardware vorgestellt. In Vorbereitung darauf wurde im Rahmen dieser Arbeit neben der Entwicklung der Komponententests eine Reihe von Kalibrierungen sowie Analog- und Digitaltests durchgeführt. Diese bilden Teile eines von Daniel Brüderle entwickelten Frameworks und werden im folgenden in ihrem Funktionsprinzip vorgestellt. Für eine ausführlichere Darstellung siehe [6].

Digitaltests

Im Rahmen dieser Tests wird die grundlegende Funktionalität des Digitalteils der Spikey-Chips verifiziert. Der *loopback test* sendet eine Anfrage ähnlich eines *pings,* der auf eine vorhandene Kommunikation mit dem Chip überprüft (siehe [14]). Die Parameter- und Synapsen-RAM Tests verifizieren, dass die entsprechenden Speicher korrekt beschrieben und wieder ausgelesen werden können. Die digitale Repräsentation eines Aktionspotenzials als Zeitangabe seiner Auslösung und seiner Zielbestimmung sind sogenannte *events* (siehe Abschnitt 1.1.2). Die korrekte Erzeugung, Erfassung und Auslieferung dieser *events* durch den Digitalteil des Chips wird mit Hilfe des *event loopback tests* überprüft (siehe [14]).

Kalibrierung der Spannungsgeneratoren

Auf dem Spikey-Chip befinden sich 46 programmierbare Spannungs-ge-ne-ra-toren einheitlicher Bauart, welche beispielsweise die Schwellspannung oder die Umkehrpotenziale der auf dem Chip realisierten Neuronen erzeugen. Dabei ist der Zusammenhang zwischen dem übergebenen Spannungswert U_{in} und der tatsächlich erzeugten Spannung U_{out} innerhalb eines Bereiches linear mit der Steigung m_{U} und dem Offset U_{off}. Steigung und Offset sowie die Spannungen, die sich an den Grenzen des linearen Bereiches einstellen, variieren für die unterschiedliche Spannungsgeneratoren. Eine Kalibrationsroutine misst daher diese Werte und speichert sie auf einem Chip [34], sodass sie bei jeder Einstellung der Spannungsgeneratoren verwendet werden können.

Analogtests

Während die Digitaltests unabhängig von den analogen Neuronen- und Synapsenschaltungen sind, umfasst der Analogtest oder auch *response test* in Form eines einfachen Integrationstests diese mit. Ein über PyNN definiertes Experiment stimuliert mit hoher exzitatorischer Intensität jedes einzelne Neuron individuell. Dann wird anhand der an den Hostcomputer zurückgegebenen Eventdaten überprüft, ob für alle Neuronen Spikes registriert werden können.

Kalibrierung der Membranzeitkonstanten

Die Membrankapazität C_m der Neuronenschaltkreise ist durch die Hardware fest vorgegeben. Durch Variationen auf Transistorebene unterscheidet sie sich von Neuron zu Neuron. Um trotzdem eine gleiche Zeitkonstante für die unstimulierten Membranen zu erzielen, kann über die Leckleitfähigkeit g_leak jeweils dieselbe Membrankonstante $\tau_\mathrm{mem} = \frac{C_\mathrm{m}}{g_\mathrm{leak}}$ eingestellt werden. Da der g_leak entsprechende Kontrollstrom $I^\mathrm{ctrl}_\mathrm{leak}$ zwar geschrieben aber nicht überprüft werden kann, misst eine Routine die resultierende Membrankonstante τ_mem und verstellt $I^\mathrm{ctrl}_\mathrm{leak}$ solange, bis der gewünschte Wert erreicht ist.

Die Ergebnisse der durchgeführten Kalibrierungen und Tests wurden in das Wiki der Electronic Visions Group eingetragen (siehe Anhang A.1), wo Protokoll über den aktuellen Status der einzelnen Spikey-Chips geführt wird.

3 Liquid Computing auf der FACETS Stage 1 Hardware

Nachdem in den vorherigen Abschnitten auf Aspekte der Voraussetzungen zur Inbetriebnahme der FACETS-Hardware eingegangen wurde, hat dieser Abschnitt die tatsächliche Emulation eines neuronalen Netzes zum Ziel. Das Modell der in Abschnitt 1.3 beschriebenen Liquid State Machine soll auf die Hardware übertragen und dort Experimente mit ihm durchgeführt werden.

Die *LSM* nach Maass, Natschläger und Markram besteht aus drei Modulen: Dem *Input*, dem *Liquid* selbst und der Ausleseeinheit. Die Ausleseeinheit kann auf sehr unterschiedliche Weise implementiert sein und ist in diesem Fall auf Grund der Verwendung eines überwachten Lernalgorithmus in Software geschrieben, wie auch der Input (siehe Abschnitt 3.6). Das *Liquid* wird im Rahmen dieses Experiments als neuronales Netz realisiert und stellt damit den Teil der *LSM* dar, der auf der FACETS-Hardware emuliert werden soll.

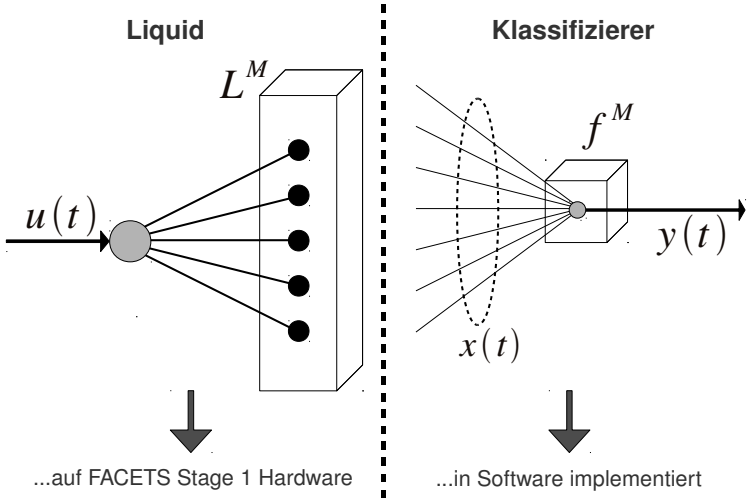

Abbildung 9: Umsetzung der Liquid State Machine

Viele verschiedene Möglichkeiten sind für die Realisierung eines Liquids denkbar. Auch wenn man sich wie in diesem Fall auf eine Implementierung als neuronales Netz festlegt, besteht weiterhin viel Freiraum hinsichtlich der Zell- und Synapsenrealisierungen.

Diese Vielfalt birgt ein großes Potenzial, sie wirft jedoch gleichzeitig die Frage nach einem sinnvollen Ausgangspunkt für ein Experiment auf der FACETS-Hardware auf. Aus diesem Grund ist es sinnvoll, im Rahmen dieser Arbeit die Realisierung der *LSM* so weit wie möglich am Beispiel in [24] zu orientieren.

3.1 Die PyNN-Beschreibung des Liquids

Die Autoren von [24] führten sämtliche Experimente auf dem Softwaresimulator CSIM durch und beschrieben diese über dessen Schnittstelle zu MATLAB [36]. Für die Umsetzung auf der neuromorphen FACETS-Hardware ist jedoch eine Beschreibung in der Metasprache PyNN nötig. Dies macht es außerdem möglich, weitere in das PyNN-Framework eingebundene Simulatoren nutzen zu können (siehe Abschnitt 1.1.3). Insbesondere kann die *LSM* auf diese Weise in NEST simuliert werden, der oft als Referenzsimulator für die FACETS-Hardware herangezogen wird.

Für die Umsetzung in PyNN wurde die Funktionalität der High-Level-API verwendet, da sie im Gegensatz zur Low-Level-API den Umgang mit einer größeren Anzahl an Neuronen bzw. Synapsen erleichtert. Grundsätzlich ist das im Rahmen dieser Arbeit entwickelte, die PyNN-API verwendende `Python`-Skript in vier Module unterteilt, deren Aufbau an dieser Stelle kurz detailliert wird:

- `column.py`: Definiert eine Klasse `column`, die eine Säule gemäß [24] innerhalb des *Liquids* repräsentiert. Sie enthält zwei Populationen `PyNN.Population` E und I von insgesamt N Integrate-and-Fire Neuronen des Typs `IF_facets_hardware1`, d.h. eine hardwarekompatible Variante eines konduktanzbasierten *Integrate-and-Fire*-Neurons. E enthält nur Neuronen, deren ausgehende Synapsen exzitatorische Wirkung besitzen (insgesamt $(1 - \text{ratio}_{inh}) \cdot N$), während I entsprechend die inhibitorisch feuernden Zellen ($\text{ratio}_{inh} \cdot N$) beinhaltet. Die Neuronen werden zufällig auf einem Gitter der Ausmaße $n_\text{x} \times n_\text{y} \times n_\text{z}$ verteilt, in dem das Attribut `PyNN.Population.position` passend gesetzt wird. Die Funktionalität für die Erstellung der Synapsen mit distanzabhängiger Verbindungswahrscheinlichkeit stellt die `PyNN` zugehörige Klasse `PyNN.DistantDependentProbabilityConnector` zur Verfügung. Dieser wird der Ausdruck $C_{ij} \cdot exp(-(\frac{d}{\lambda})^2)$ übergeben, wobei C_{ij} mit $i, j \in \{\text{E,I}\}$ die Verbindungswahrscheinlichkeit zwischen pre- und postsynaptischem Neuron linear skaliert und λ die Distanzabhängigkeit bzw. Reichweite charakterisiert. Des weiteren werden synaptische Gewichte w_{ij} mit $i, j \in \{\text{E,I}\}$ übergeben.

- `liquid.py`: Definiert eine Klasse `liquid`, die aus einer bis mehreren Instanzen von `column` besteht. Des weiteren enthält `liquid` eine `PyNN.population` der Größe N_input, dessen Neuronen des Typs `PyNN.SpikeSourceArray` mit je einem zufällig ausgewählten Neuron der einzelnen Instanzen von `column` exzitatorisch verbunden sind.

- `main.py`: In diesem Modul wird eine Klasse `maassLiquid` definiert, die eine Instanz von `liquid` beinhaltet und diese mit dem *Input* verbindet. Hier werden die Befehle `PyNN.setup()` und `PyNN.run()` aufgerufen.

- `config.py`: Enthält die Werte für alle oben beschriebenen Parameter

Letztendlich spiegelt die Verwendung der High-Level-API die Tatsache wieder, dass die genaue Implementierung der synaptischen Rückkopplung, bzw. der genaue Verlauf der Synapsen im Rahmen der LSM keine Rolle spielt. Vielmehr müssen statistische Parameter gefunden werden, die eine Lösung der gegebenen Probleme erlauben.

3.2 Hardwarespezifische Modifikationen

Während einige Elemente des in [24] beschriebenen Netzwerks in die Sprache der von der Hardware unterstützten Modelle übersetzt werden können, müssen manche Aspekte verworfen und gegebenenfalls durch Modifikationen an anderer Stelle kompensiert werden.

Nicht unterstützt werden beispielsweise die synaptischen Verzögerungen (*synaptic delays*). Für sie gibt es keine sinnvolle Entsprechung auf der Hardware, daher können sie im folgenden nicht modelliert werden. Zur Zeit der Durchführung des Experiments war die Einstellung der Refraktärzeit ebenfalls nicht möglich.

Des weiteren beträgt in [24] die Potenzialdifferenz zwischen Ruhepotenzial und der Feuerschwelle $\Delta U = 1,5\,\text{mV}$. Dieser Wert liegt in der Größenordnung der Hardwareschwankungen und wird aus diesem Grund höher gewählt. Da die n_input Eingangskanäle auf jeweils nur ein Neuron des *Liquids* projiziert werden, müssen die Zellen jedoch trotzdem eine hohe Sensibilität aufweisen. Die Gewichte der Synapsen zwischen der Input-Population und denen des *Liquids*

wurden auf den Maximalwert von $w_{\text{input}} = 15 \cdot g_{\max}$ eingestellt, da kleinere Werte nur sehr wenig Aktivität bewirken konnten. Gleichzeitig wurde der Übersetzungsfaktor zwischen den biologischen und den digitalen 4-Bit Gewichten auf den maximalen Wert festgesetzt. Anschließend wurden das Ruhepotenzial E_{rest}, das Rückstellpotenzial E_{reset} und die Feuerschwelle V_{thresh} so gesetzt, dass eine Poisson-verteilte Stimulation der Frequenz $v_{\text{inp}} = 80\,\text{Hz}$ Aktionspotenziale mit einer durchschnittlichen Frequenz von ca. $v_{\text{fire}} = 10\,\text{Hz}$ hervorrief. Diese Werte sind in biologischen Zeiteinheiten angegeben und entsprechen einem biologisch realistischen Verhalten [2].

Weiterhin wird in [24] zu Anfang jedes Experiments das Membranpotenzial jeder Zelle mit einem zufälligen Wert initialisiert. Dies ist so auf der Hardware nicht möglich. Die Experimente mit der *LSM* umfassen jedoch eine gewisse Anlaufzeit (siehe Abschnitt 3.4), wobei davon ausgegangen wird, dass die Summe der in dieser Phase auftretenden Hardwareschwankungen qualitativ eine der zufälligen Initialisierung der Membranpotenziale ähnliche Auswirkung auf das Experiment besitzt.

Dynamische Synapsen bzw. die *Short Term Plasticity* (STP) ist eine Funktionalität, die von der Hardware nur eingeschränkt angeboten wird. Maass et al. modellieren je nach Typ von pre- und postsynaptischem Neuron einer Synapse das dynamische Verhalten unterschiedlich. Die Hardware wies diesbezüglich zwei unterschiedliche Einschränkungen auf. Zum einen kann grundsätzlich für einen Synapsentreiber, der für die Synapsen eines einzelnen presynaptischen Neurons verantwortlich ist, jeweils nur eine Parametereinstellung für den STP-Mechanismus vorgenommen werden. Zum anderen besteht die zweite Einschränkung darin, dass zum Zeitpunkt der Durchführung die Synapsen nur entweder als *depressing* oder *facilitating* eingestellt werden konnte. Beides gleichzeitig funktionierte nicht.

Nach Maass et al. verbessert die STP die Berechnungsfähigkeit der *LSM*. Zusätzlich dazu übernimmt sie jedoch noch eine weitere Funktion: Ein rekurrentes Netz ist zu jeder Zeit der Gefahr ausgesetzt, für einen bestimmten Input die eingespeisten Aktionspotenziale so rückzukoppeln, dass sie sich stark potenzieren und somit das Netz seine Aktivität ungebremst erhöht. Die STP kann diesem Effekt bei geeigneter Anwendung entgegenwirken und so zur Stabilität des Netzes beitragen.

Unter dem gegebenen Umstand, dass entweder *depression* oder *facilitation* modelliert werden kann, bestünde eine Realisierungsoption darin, die Synapsen von Population E auf sich selbst dämpfend, die Synapsen von E nach I verstärkend, von I nach I dämpfend und von I nach E wiederum verstärkend zu modellieren. Auf diese Weise würde man erwarten, eine Explosion der Aktivität verhindern zu können. Auf Grund der oben genannten Hardwareeinschränkung ist diese hinsichtlich der Aktivitätskontrolle optimale Konfiguration jedoch nicht möglich. Geht man davon aus, dass die Synapsen von E nach E auf jeden Fall dämpfend realisiert werden müssen, sind dadurch ebenfalls die Verbindungen von E nach I dämpfend, was einer Aktivitätsminderung entgegenwirkt.

Unter diesen Umständen wurde es bei der Umsetzung der LSM auf Hardware vorgezogen, für die synaptischen Gewichte w_{ij} und Verbindungswahrscheinlichkeiten C_{ij} zwischen den Populationen i und j von den Werten in [24] abzuweichen um auf diese Weise die Selbstregulierung des Netzes zu verstärken. Es wurde dazu eine Einstellung gesucht, die zum einen eine gute Leistung des Liquids ermöglicht und zum anderen einer Aktivität entspricht, die einigen Freiraum hin zu höheren mittleren Feuerraten aufzeigt (siehe Abschnitt 3.5).

Für eine Tabelle der im Rahmen dieser Arbeit verwendeten Parameter siehe Anhang A.2. Die nicht in diesem Abschnitt erwähnten Größen entsprechen den in [24] gewählten Einstellungen. Für alle restlichen Parameter galten die voreingestellten Standardwerte von PyNN. Ferner entsprechen im folgenden alle Zeitangaben der biologischen Interpretation der tatsächlichen Emulationszeiten auf der Hardware, die über den Beschleunigungsfaktor 10^4 zu ermitteln sind.

3.3 Der Klassifizierer in Software

Während das Liquid vollständig auf der FACETS Stage 1 Hardware realisiert werden konnte, wurde der lineare Klassifizierer (siehe Abschnitt 1.3) in Software implementiert. Grundsätzlich können IF-Neuronen als Klassifizierer verwendet werden mit der Einschränkungen, dass die FACETS Stage 1 Hardware pro Neuron ohne aufwendige Modifikationen nur 16 verschiedene Gewichtswerte unterstützt. Im Rahmen der hier präsentierten Studien wird der Zustand des Liquids vorerst an Software übertragen und dort weiterverarbeitet. Zu einem späteren Zeitpunkt wäre es jedoch durchaus möglich, den Klassifizierer ebenfalls auf die FACETS Stage 1 Hardware abzubilden.

Es wurde ein Klassifizierer implementiert, der auf der Funktionsweise eines *Perzeptrons* basiert. Aus den Zeitpunkten, zu denen im Liquid die Aktionspotenziale ausgelöst wurden, berechnet sich ein Zustandsvektor. Die Komponenten dieses Vektors stellen den Feuerverlauf der Liquidneuronen als Skalare dar, wobei die einzelnen Spikezeiten einer Faltung unterworfen werden. Der vom Perzeptron ausgelesene Zustandsvektor $x(t)$ des Liquids zu einem Zeitpunkt t wird durch folgende Formel bestimmt:

$$\vec{x}(t) = \sum_{n=1}^{N} \sum_{i=1}^{I_n} \theta(t - t_{n_i}) exp(-(t - t_{n_i})/\tau) \cdot \vec{e}_n \,, \tag{3}$$

wobei N die Anzahl der Neuronen im Liquid, I_n die Gesamtzahl der Spikes vom Neuron mit Index n, t_{n_i} der Zeitpunkt dessen i-ten ausgelösten Spikes, τ die Faltungskonstante und e_n den n-ten Einheitsvektor darstellen. Dieser Zustandsvektor wird mit einem Gewichtsvektor $\vec{w} = \sum_{n=1}^{N} w_n \vec{e}_n$ multipliziert. Das Perzeptron wandelt dieses Produkt $p = \vec{x} \cdot \vec{w}$ anschließend in einen binären Wert y um, der abhängig vom Vergleich mit einem Schwellwert p_S ist:

$$y = \begin{cases} 1 & p = \vec{x} \cdot \vec{w} > p_S \\ 0 & p = \vec{x} \cdot \vec{w} < p_S \end{cases} \,. \tag{4}$$

Der Output$y$ kann damit binäre Aussagen über den Input treffen. Der Lernalgorithmus modifiziert während dem Training die w_n um eine Lernschrittweite l, gewichtet durch die Zustandswerte x_n. Die Anfangsschrittweite l_0 fällt dabei exponentiell mit der Anzahl der Lernschritte und der Konstanten τ_relax ab.

Die Faltungskonstante τ aus Glg. 3 ist ein Maß dafür, wie weit dem Klassifizierer der Rückblick in die Vergangenheit erlaubt wird. Ein wie in Abschnitt 1.3 beschriebener Klassifizierer ohne Erinnerung wäre möglich, wenn man als Zustand die anliegenden Membranpotenziale betrachten würde. Im Fall der Realisierung auf Hardware ist dies jedoch nicht möglich. Stattdessen werden die Spikezeiten als Informationsträger verwendet. Daher muss ein Intervall festgelegt werden, in diesem Fall charakterisiert durch τ, das die Reichweite der Berücksichtigung von Spikes im Liquid festlegt.

3.4 Methode zur Evaluierung der Leistung der LSM

Um die Leistung der implementierten LSM beurteilen zu können, wurde eine in [24] vorgeschlagene Klassifizierungsaufgabe verwendet. Dabei dienen zwei zufällig generierte, der Poisson-Statistik[17] folgende Spikemuster A und B der zeitlichen Länge T mit N_input Inputkanälen als Vorlagen für die zu präsentierenden Trainingseingaben. Diese Vorlagen werden jeweils in Einzelstücke der

[17] Die Spikezeiten für die Muster A und B entstehen für jedes Neuron jeweils aus einem Poissonprozess mit gleicher Frequenz.

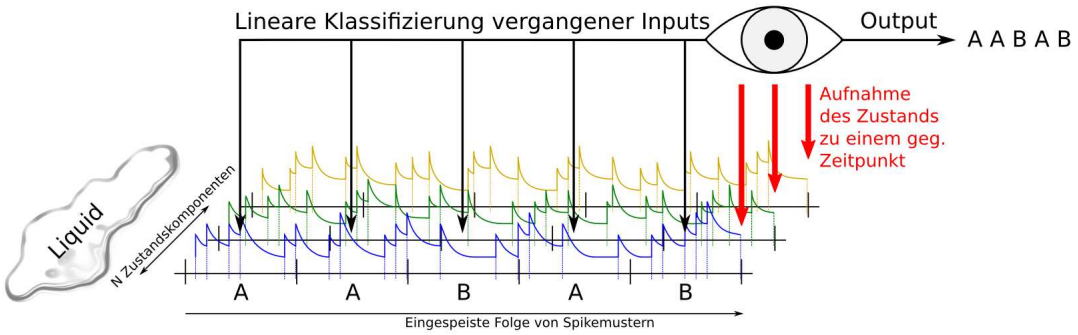

Abbildung 10: Veranschaulichung der Funktionsweise des Klassifizierers: Der Klassifizierer liest das Ergebnis der Faltung (siehe Abschnitt 3.3) zu einem geg. Zeitpunkt ab und produziert einen binären Output. Für die Erklärung der Spikemuster A und B siehe Abschnitt 3.4. Abbildung mit freundlicher Erlaubnis von Sebastian Jeltsch.

Länge $\triangle T$ unterteilt (siehe Abbildung 11). Anschließend werden Spikemuster C_i zufällig aus diesen Einzelstücken zusammengestellt: Für jedes Zeitintervall Δt wird entweder das Muster von A oder von B übernommen. Zusätzlich dazu werden die einzelnen Spikezeiten um zufällige Werte gestreut, die einer Gaußverteilung der Breite σ entnommen sind.

Die Spikemuster C_i werden in das Liquid eingespeist. Der Klassifizierer wird nun darauf trainiert, auszugeben, ob für das gegebene C_i für verschiedene Zeitintervalle die Inputs von Spikemuster A oder B angelegen haben. Dafür werden die folgenden Perzeptrone verwendet:

- Erstes Perzeptron: Zeitintervall $0 < t < \Delta T$
- Zweites Perzeptron: Zeitintervall $\Delta T < t < 2\Delta T$
- i-tes Perzeptron: Zeitintervall $(i-1)\Delta T < t < i\Delta T$

Die Perzeptrone müssen also hierbei Aussagen über bereits vergangene Inputs treffen. Man erwartet, dass bspw. das dritte Perzeptron eine schwierigere Klassifierzungsaufgabe besitzt als das erste, da dieses nicht auf das Liquid als Speicher angewiesen ist.

Für die im Rahmen dieser Arbeit durchgeführten Experimente wurde $T = 500\,\text{ms}$ und $\Delta T = 50\,\text{ms}$ gewählt. Die Funktionsweise der LSM gibt diese Zeiten nicht zwingend vor. Da die Membranzeitkonstanten der verwendeten Liquidneuronen bei ca. $\tau_{\text{mem}} = 30\,\text{ms}$ lagen, ist es sinnvoll, die Länge der betrachteten Spikemuster im Vergleich dazu etwas größer zu wählen. Ein mögliches Erinnerungsvermögen des Liquids durch die Integrationseigenschaft der Membranen beschränkt sich somit auf ein Zeitintervall. Ein weiterer, eher biologisch motivierter Grund besteht darin, dass Effekte der synaptischen Langzeitplastizität *STDP* (Spike Time Dependent Plasticity) ähnliche Zeitkonstanten aufweisen [32]. So verändern z.B. zwei Spikes die Gewichte einer Synapse, wenn ihr zeitlicher Abstand nicht mehr als $\Delta t \approx 50\,\text{ms}$ beträgt. Es scheint dies also eine für auf Spikezeiten basierendes Erinnerungsvermögen relevante Größenordnung zu sein. Die Gesamtdauer der Experimente wurde über die drei relevanten Zeitintervalle hinaus gewählt, um eine ausreichende Initialisierung des Liquids zu bewirken.

Für die gesamte Experimentdauer von 500ms sind $2^{10} = 1024$ verschiedene C_i bzw. Inputkombinationen möglich. Betrachtet man das dritte Perzeptron, das auf das dritte Spikemuster in der Vergangenheit trainiert ist, würden $2^7 = 128$ vorherige Kombinationen als Initialisierungen möglich sein. Weitere $2^2 = 4$ verschiedene Spikemuster würden das Liquid bis

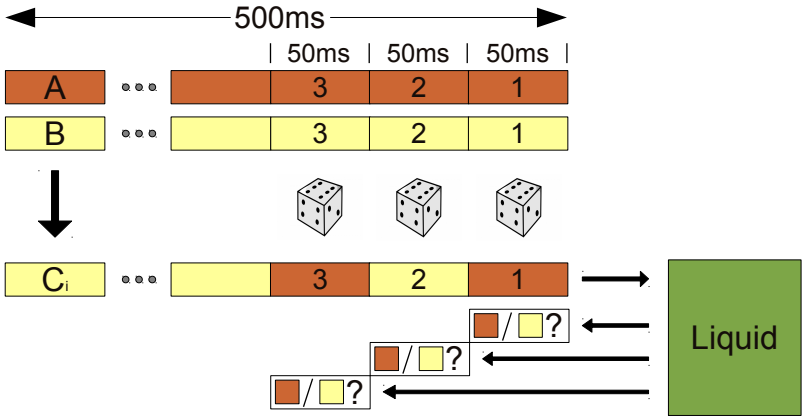

Abbildung 11: Die Spikefolgen A und B werden in Teilstücke der Länge $\Delta T = 50\,\mathrm{ms}$ zerschnitten. Neue Spikefolgen C_i werden erstellt, in dem für jedes Zeitintervall zufällig entweder die Vorlage von A oder B verwendet wird. Die Perzeptrone 1,2 und 3 werden auf die Bestimmung der Herkunft der Teilstücke trainiert.

zum Auslesen des Zustands beeinflussen und verzerren. Wollte man also, dass dem Klassifizierer während dem Training bereits jedes mögliche Szenario präsentiert wird (abgesehen von der Streuung der Spikezeiten, siehe oben), müsste man diesen auf 1024 verschiedene Kombinationen trainieren. Dadurch würde man jedoch die Information verlieren, in wie weit der Klassifizierer eine Generalisierung der ihm präsentierten Zustände vornimmt.

Aus diesem Grund wurden insgesamt $N_{\mathrm{samples}} = 100$ zufällige Spikemuster C_i aus den Vorlagen A und B erstellt. Davon dienten $N_{\mathrm{training}} = 60$ für das Training und die restlichen $N_{\mathrm{test}} = 40$ für das Testen des Klassifizierers. Um das Training effektiver zu machen, wurden, wenn nicht anders angegeben, die Trainingsmuster in fünf Wiederholungen gelernt. Dies erlaubte die Wahl kleinerer Lernraten, die für die Konvergenz des Lernalgorithmus entscheidend sind. Die Anfangslernrate lag bei $l_0 = 0,1$ und fiel mit $\tau_{\mathrm{relax}} = 200$ ab. Der von N_{test} richtig klassifizierte Prozentsatz stellt die Leistung der LSM für die gegebenen Einstellungen und Perzeptrone dar und wird im folgenden als die Richtigkeit oder Korrektheit des i-ten Perzeptrons bezeichnet. Für die Bestimmung wurde jeweils über 5 Klassifizierungen gemittelt. Die angegebene Ungenauigkeit entspricht dem Fehler des Mittelwerts der Mittelung über zehn Liquids unterschiedlicher Anfangszwerte des Zufallszahlengenerators.

3.5 Beobachtung verschiedener Dynamikbereiche

Wie in Abschnitt 1.3 beschrieben erwartet man, dass das Liquid zwei charakteristische Verhaltensweisen aufzeigt. Auf der einen Seite gibt es den Dynamikbereich, in dem die Aktivität größtenteils vom Input gesteuert wird. In diesem Fall spiegelt das Liquid näherungsweise den Input wider. Auf der anderen Seite erwartet man ein Regime, das von sehr hoher Aktivität geprägt ist. Informationen über den Input sind hier nur noch sehr schwer zu extrahieren. In [3] wird gezeigt, dass es sich dabei um chaotisches Verhalten handeln kann. Da darauf an dieser Stelle nicht weiter eingegangen werden soll, wird das beschriebene Dynamikregime im folgenden als *turbulent* bezeichnet.

Zwei ausschlaggebende Parameter, die über die Dynamik des Liquids entscheiden, sind die Inputeigenschaften und die Konnektivität des Liquids. Damit hängt von ihnen auch die Klassifizierungsleistung der LSM ab. Die relevanten Inputeigenschaften sind in diesem Fall die Frequenz der zufällig generierten Spikezeiten, die Anzahl der Neurone, auf die im Liquid projiziert wird und die hierbei verwendeten Synapsengewichte. Die Konnektivität des Liquids ist durch die synaptischen Verbindungen gegeben: Ihr Typ, ihre Dichte und ihre Gewichte spielen dabei eine Rolle.

Es hat sich in Vorstudien zu den präsentierten Experimenten jedoch gezeigt, dass das Verhalten des Liquids nicht nur auf diese Parameter sehr sensibel reagiert, sondern auch spontan seine Dynamik radikal verändern kann. Dies ist durch die zufällige Natur der eingespeisten Inputs bedingt, die das Netzwerk trotz gleicher mittlerer Frequenz auf unterschiedliche Weise anregen können. Eine anschauliche Erklärung besteht darin, dass in einem gegebenen Zeitfenster bspw. kurzzeitig überdurchschnittlich mehr exzitatorische als inhibitorische Neuronen stimuliert werden, was zu einem nicht mehr zu bremsenden Anstieg der allgemeinen Aktivität führen kann.

Die Abbildungen 12, 13 und 14 sind sogenannte *Rasterplots*. Sie stellen die Spikezeiten während der gesamten Experimentdauer für alle beteiligten Neuronen als Punkte dar. Die obersten 40 Zeilen repräsentieren den Input, während die unteren 135 die Spikes innerhalb des Liquids darstellen. Anhand dieser Abbildungen soll der oben beschriebene spontane Wechsel in der Netzwerkdynamik exemplarisch dargestellt werden. Die Bildlegenden erhalten die jeweiligen Erklärungen.

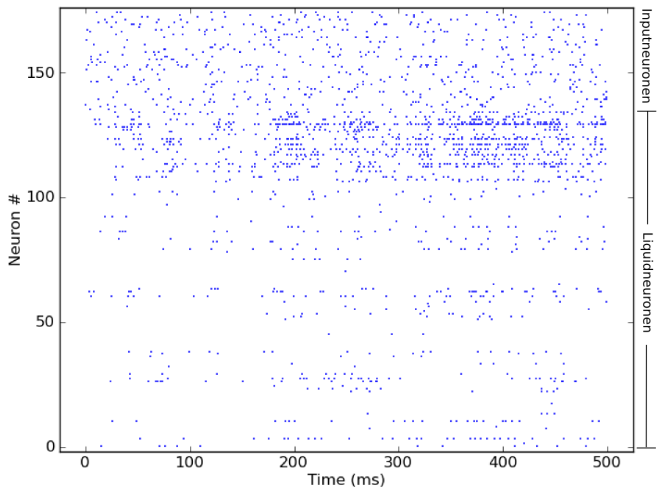

Abbildung 12: Im unteren Bereich zeigt das Netzwerk zwar nur eine moderate Aktivität, doch die obersten Liquidneuronen weisen bereits eine hohe Dynamik auf. Vereinzelt sind Neuronen zu erkennen, die innerhalb eines kurzen Zeitraums mit sehr hoher Frequenz feuern.

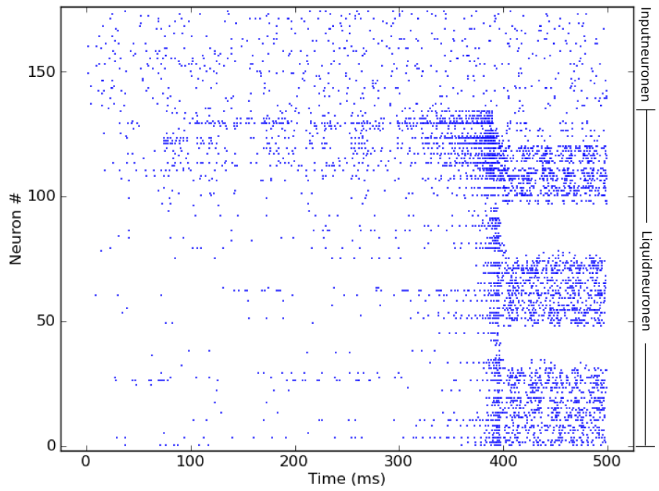

Abbildung 13: Hier tritt bis $t \approx 400\,\mathrm{ms}$ biologischer Zeit ein zum Experiment in Abb. 12 sehr ähnliches Verhalten auf. Ab $t \approx 400\,\mathrm{ms}$ ist der in Abschnitt 3.5 erwähnte spontane Übergang von dem zuvor beobachteten Verhalten in ein Regime deutlich höherer Aktivität zu sehen. Die Bereiche bzw. Neurone, die in dieser Phase hoher Aktivität nicht feuern, werden aus technischen Gründen nicht ausgelesen. Über ihr tatsächliches Verhalten kann hier also keine Aussage getroffen werden.

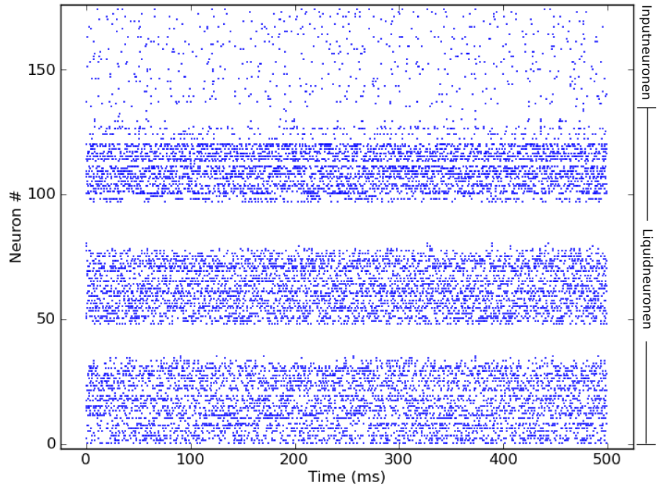

Abbildung 14: Für identische Parameter zu den Exp. aus Abb. 12 und 13 zeigt dieses Exp. bei Input gleicher Frequenz im Vergleich zum Input viel größere Aktivität. Das sich in Abb. 13 bei $t \approx 400\,\mathrm{ms}$ ankündigende Verhalten ist hier über die gesamte Dauer des Experiments zu beobachten. Bezüglich der Bereiche scheinbar fehlender Aktivität siehe die Bildunterschrift von Abb. 13.

Bei den Zeitangaben ist jeweils zu beachten, dass diese in ihrer biologischen Entsprechung angegeben sind. Um die tatsächliche Zeit der Vorgänge auf dem Chip zu erhalten, müssen obige Werte mit dem Beschleunigungsfaktor 10^4 umgerechnet werden. Zehn biologische ms entsprechen einer μs auf der FACETS Stage 1 Hardware.

Die Betrachtung von Abbildungen 13 und 14 lässt scheinbar darauf schließen, dass die Neuronen jeweils um Neuronenindex 40 und 80 innerhalb der Phase hoher Aktivität nicht feuern. Dem ist jedoch nicht so, da sich an dieser Stelle eine Bandbreitenbegrenzung bemerkbar macht. Ab einer bestimmten Anzahl von Ereignissen, die vom Digitalteil des Chips verarbeitet werden müssen, werden auf Grund einer technischen Limitierung nicht alle Spikes an den Hostcomputer zurückgegeben. Die Verlangsamung der Emulierung von einem Beschleunigungsfaktor 10^5 hin zu den oben genannten 10^4 durch eine Chip-Revision weitete den für Experimente zugänglichen Feuerratenbereich bereits erheblich aus. Trotzdem stößt das Liquid teilweise an diese Grenzen. Für eine genaue Behandlung der Bandbreiteneinschränkung siehe Kapitel 4 in [6].

Diese hohe Sensibilität des Liquids mit abrupten Auswirkungen auf die mittlere Feuerrate führte dazu, dass im Rahmen dieser Arbeit eine sehr hohe Inhibition innerhalb des Netzwerks gewählt wurde. Wie in Abschnitt 3.2 erwähnt, wurden für die Einstellungen der synaptischen Verbindungswahrscheinlichkeiten und der Synapsengewichte Werte gewählt, die das Netzwerk hinsichtlich seiner Aktivität kontrollieren. Für die in dieser Arbeit präsentierten Experimente wurden die in Anhang A.2 aufgeführten Parameter verwendet. Dabei soll kein Anspruch auf die optimalste Einstellung gehoben werden. Zu ihrer Wahl führte das Kriterium, einen dynamischen Bereich ausfindig zu machen, der eine möglichst große Variation des Konnektivitätsparameters λ erlaubte. Da eine systematische Erkundung des gesamten Parameterraumes im Rahmen dieser Arbeit nicht möglich gewesen wäre, wurde von den in [24] angegebenen Werten ausgegangen, um schrittweise Erfahrungen über das Verhalten des Liquids zu sammeln und letztendlich einen sinnvollen Arbeitspunkt zu finden.

3.6 Selektive Parameterstudien

Im folgenden Abschnitt geht es darum, mit Hilfe der in Abschnitt 3.4 beschriebenen Methode die Klassifizierungsleistung der LSM zu analysieren. Dabei wird die Klassifizierungskorrektheit der einzelnen Perzeptrone betrachtet, die Aussagen über unterschiedlich weit in der Vergangenheit liegenden Input treffen.

Insbesondere stellt sich die Frage, wie viel der Klassifizierungsleistung jeweils dem Speicher des Liquids selbst zuzusprechen ist und wieviel dem Erinnerungsvermögen des Klassifizierers. Dieses ist durch die Faltungskonstante τ gegeben. Nach Glg. 3 ist das die Zeit, nach der ein Spike nur noch einen Bruchteil $\frac{1}{e}$ seines ursprünglichen Einflusses auf die entsprechende Zustandskomponente seines Neurons besitzt. Ein verschwindendes τ würde einen Klassifizierer ohne Erinnerung bedeuten, in diesem Fall würde jedoch kein einziger Spike in das Ergebnis einfließen. Ein kleineres τ bedeutet also ein kleineres Erinnerungsvermögen, gleichzeitig jedoch die Nicht-Berücksichtigung vieler Spikes, die nicht zur Zeit des zu bestimmenden Inputmusters vorgekommen sein müssen. Trotzdem gilt: Je kleiner τ, desto wahrscheinlicher ist die Klassifizierungsleistung dem Liquid zuzusprechen.

3.6.1 Konnektivitätsparameter λ und Erinnerung des Klassifizierers τ

Im folgenden wird die Leistung des Liquids in Abhängigkeit des Konnektivitätsparameters λ (siehe Abschnitt 3.1) untersucht und diskutiert. Dieser ist ein Maß für die räumliche Reichweite der vorkommenden Synapsen. Dadurch vergrößert sich mit ihm ebenfalls die Gesamtzahl der Synapsen des Netzwerks, womit λ den Grad der Rückkopplung des Liquids angibt. Ein sehr

kleiner Wert beschreibt Einstellungen, in welchen der Input fast ausschließlich im Netzwerk gespiegelt wird, größere Werte stehen dagegen für eine regere Rückkopplungsdynamik. Um Zufallseffekte möglichst auszuschließen, wurde für die folgenden Messungen über zehn verschiedene Anfangswerte der Zufallsgeneratoren gemittelt, wobei der angegebene Fehler dem Fehler des Mittelwerts entspricht.

Abbildung 15 zeigt die Korrektheit des ersten Perzeptrons, also die Bestimmung des zuletzt angelegenen Input. Bei $\tau = 3\,\text{ms}$ wurde diese Messung für verschiedene Werte der Konnektivität λ durchgeführt. Die Verbindungslinie soll dabei wie auch in folgenden Abbildungen nur die Zusammengehörigkeit der Punkte kennzeichnen. Da in diesem Fall zwischen dem letzten Anliegen des zu bestimmenden Musters und der Klassifizierung selbst keine Zeit liegt, entsprechen die 100% Korrektheit für kleine λ den Erwartungen. Ab einer Konnektivität von ca. $\lambda = 1,8$ nimmt die Erkennungsfähigkeit jedoch rapide ab. Dies lässt sich durch den Übergang des Liquids in den turbulenten Bereich (ab sofort mit TB bezeichnet) erklären. Offensichtlich geht hier Information über den gegenwärtigen Input bereits verloren.

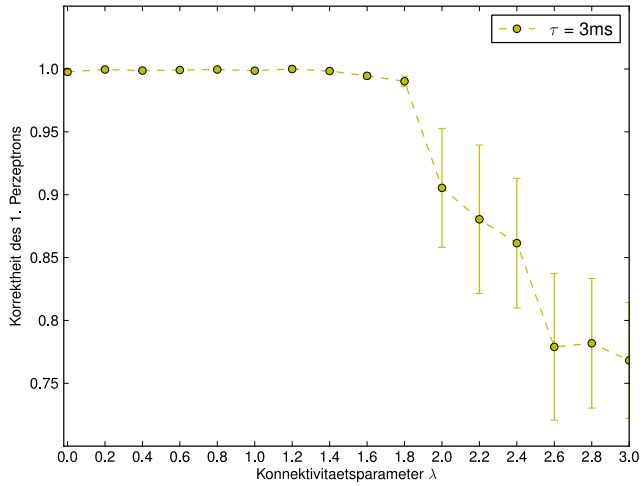

Abbildung 15: Korrektheit des ersten Perzeptrons, das auf das letzte Zeitintervall trainiert ist. Für kleine λ ist erwartungsgemäß die Erkennung des momentanen Inputs sehr gut, da bei geringer Konnektivität das Liquid vom Input dominiert wird. Ab ca. $\lambda = 1,8$ sinkt die Klassifizierungsleistung rasant, was auf das Eintreten des turbulenten Zustands zurückzuführen ist.

Abbildung 16 zeigt für die gleiche Erinnerungstiefe τ des Klassifizierers das Ergebnis des zweiten Perzeptrons. Diese Aufgabe ist schwieriger als die vorherige, da in diesem Fall Rückschlüsse über die Vergangenheit zwischen 50 ms und 100 ms gezogen werden. Für kleine λ ist zwar schon ein von 50% verschiedenes Ergebnis zu erkennen (was das Fehlen von jeglicher Klassifizierungsfähigkeit widerspiegeln würde), jedoch gibt es gegen größere λ einen Anstieg. Dies lässt darauf schließen, dass in diesem Fall offensichtlich die Konnektivität des Liquids die Ursache für ein längeres Gedächtnis ist. Für diese Einstellung macht sich also der sogenannte schwindende Speicher des Liquids bemerkbar. Nachdem bei $\lambda \approx 1,2$ ein Maximum zu beobachten ist, nimmt das Ergebnis der Klassifizierung für größer werdende λ wieder ab. Wiederum ist dies auf das Übergehen in den TB zurückzuführen, in dem jegliche Information verloren geht. Ein ähnlicher Zusammenhang zwischen der Klassifizierungsfähigkeit und der Konnektivität λ ergibt sich in den

Softwaresimulationen in [24].

Um die Auswirkungen von τ zu analysieren, enthält Abbildung 17 das Ergebnis des 2. Perzeptrons für ein sehr kleines $\tau = 1\,\text{ms}$ und für ein etwas größeres $\tau = 10\,\text{ms}$. Für das kleine $\tau = 3\,\text{ms}$ ist nur eine sehr schwache Klassifizierungskorrektheit zu erkennen, wobei es ein leichtes Maximum für $\lambda \approx 1,2$ gibt. Im Fall von $\tau = 10\,\text{ms}$ ist für kleine λ bereits eine hohe Richtigkeit zu beobachten. Dies lässt darauf schließen, dass hier das Erinnerungsvermögen des Klassifizieres eine große Rolle spielt. Wiederum ist für große Werte der Konnektivität eine sehr geringe Klassifizierungsleistung zu erkennen, ein weiteres Mal bedingt durch den zum Vorschein tretenden TB.

Die Untersuchung der Erkennungsrate des Liquids für verschiedene Konnektivitäten λ zeigt, dass für kleine und große Werte kein gutes Ergebnis zu erzielen ist. Auf der einen Seite ist für kleine λ das Erinnerungsvermögen durch die schwache Rückkopplung des Netzwerkes stark begrenzt. Für große λ hingegen kommt das turbulente Verhalten zum Vorschein. Im Übergangsgebiet zwischen diesen beiden Dynamikregimes wurde eine optimale Einstellung bei ca. $\lambda = 1,2$ ermittelt.

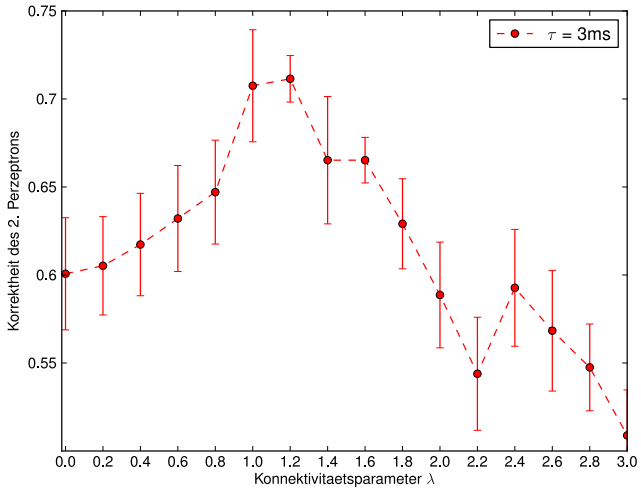

Abbildung 16: Korrektheit des 2. Perzeptrons für ein mittleres $\tau = 3\,\text{ms}$. Für sehr geringe λ ist die Klassifizierungsleistung erwartungsgemäß sehr gering, da das Netzwerk dort vom Input dominiert wird. Für große λ macht sich der turbulente Zustand bemerkbar. Dazwischen, bei $\lambda = 1,2$, gibt es ein Maximum, das die Speicherfähigkeit des Liquids widerspiegelt.

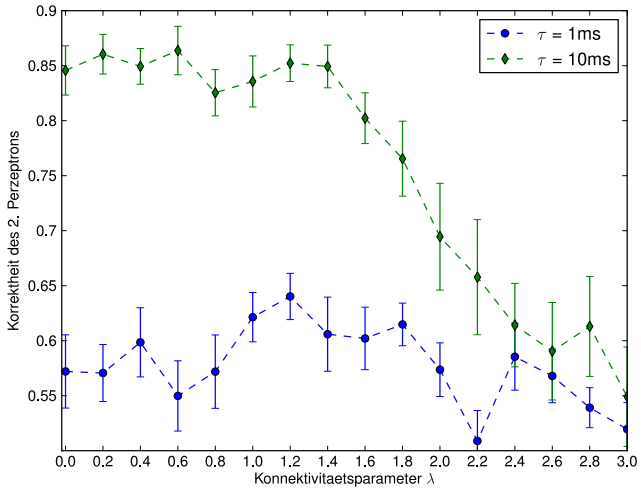

Abbildung 17: Korrektheit des 2. Perzeptrons, das auf das Zeitintervall von 100 ms bis 50 ms in der Vergangenheit trainiert ist. Für ein sehr kleines $\tau = 1\,\text{ms}$ ist die Leistung konstant gering, ein schwaches Maximum ist bei $\lambda = 1,2$ zu erkennen. Für $\tau = 10\,\text{ms}$ ist für kleine λ eine hohe Korrektheit zu sehen, da das Gedächtnis des Klassifizierers schon sehr groß ist. Gegen größere λ gibt es für beide Werte von τ einen Abfall, bedingt durch den turbulenten Zustand.

3.6.2 Klassifizierung mit und ohne Liquid

Nachdem das Klassifizierungsvermögen in Abhängigkeit der Konnektivität λ gemessen wurde, wird im Rahmen der folgenden Betrachtung das Liquid für seinen besten Wert bei $\lambda = 1,2$ verwendet. In Anbetracht der Tatsache, dass der Klassifizierer selbst ein Erinnerungsvermögen besitzt, stellt sich die Frage, wie gut seine Korrektheit bei Anwendung auf den reinen Input selbst im Vergleich zur vollständigen LSM ausfällt. Abbildung 18 zeigt diesen Vergleich für einen großen Bereich von τ unter Betrachtung des 2. Perzeptrons.

Der Klassifizierer ohne das Liquid weist für kleine τ erwartungsgemäß gar keine Klassifizierungsfähigkeit auf. Erst ab einem $\tau \approx 10\,\text{ms}$ steigt die Erkennungsrate rasant an. Im Vergleich dazu zeigt die vollständige LSM in einem sehr großen Bereich von kleinen τ bis hin zu $\tau \approx 16\,\text{ms}$ eine deutlich bessere Leistung als mit nur ungefiltertem Input. Der auch in diesem Experiment zu beobachtende schwindende Speicher des Liquids bestätigt damit nochmals die korrekte Funktionsweise der LSM auf der FACETS Stage 1 Hardware.

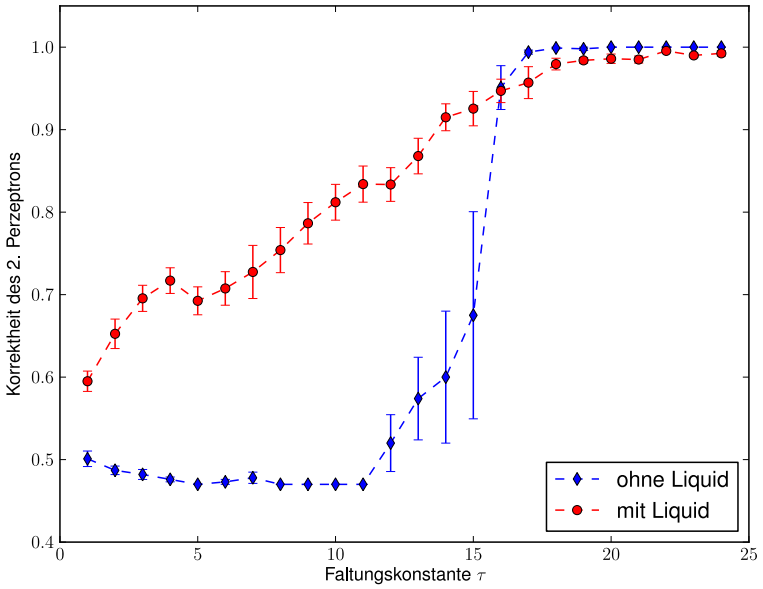

Abbildung 18: Der Klassifizierer direkt auf den Input angewandt (blau) im Vergleich zu der vollständigen LSM. Für einen sehr großen Bereich von τ ist ein deutlicher Mehrwert des Liquids zu erkennen.

3.6.3 Speicherkapazität

Abbildung 19 zeigt die Erkennungsraten der ersten fünf Perzeptrone für einen großen Bereich von τ. Anstelle von zehn verschiedenen wurde für diese Messung ein einziges Liquid für einen festen Anfangswert des Zufallszahlengenerator verwendet. Der Klassifizierer wurde in diesem Fall auf $N_{\text{training}} = 6000$ Inputs trainiert und anschließend auf $N_{\text{test}} = 4000$ Inputs getestet. Entsprechend wurde der Abfall der Lernrate auf $\tau_{\text{relax}} = 3000$ erhöht.

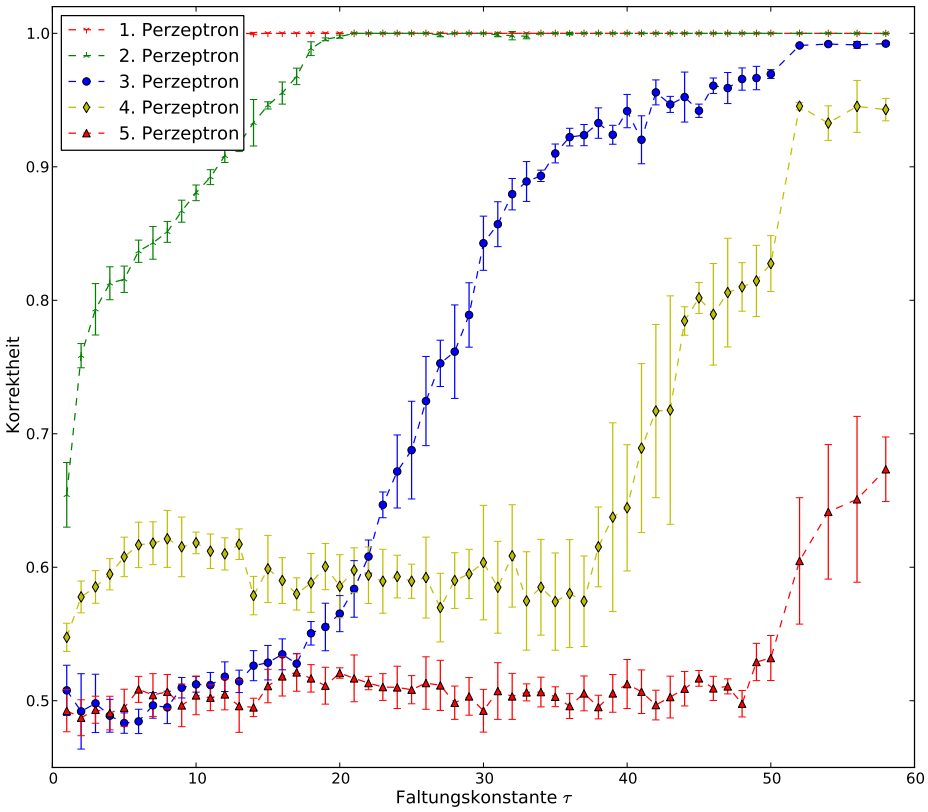

Abbildung 19: Die Erkennungsraten für die ersten fünf Perzeptrone bei unterschiedlichen Werten von τ

Die bereits bekannten Perzeptrone 1 und 2 erreichen sehr schnell die maximale Erkennungsrate. Für das dritte Perzeptron ist ab $\tau \approx 10\,\text{ms}$ ein schneller Anstieg zu beobachten, bevor sich ab ca. $\tau = 30\,\text{ms}$ die obere Grenze der Erkennungsrate bemerkbar macht. Die letzten beiden Perzeptrone zeigen erst für sehr viel größere Werte von τ einen starken Anstieg. Perzeptron 4 zeigt für sehr kleine τ einen schwachen Anstieg der Erkennungsrate, die bis $\tau \approx 20\,\text{ms}$ sogar höhere Werte als die des 3. Perzeptrons annimmt. Eine Erklärung dieses Phänomens kann in dieser Arbeit nicht gegeben werden, möglicherweise handelt es sich um eine Eigenart des für diese Messung zufällig gewählten Liquids. Des weiteren ist zu erkennen, dass die Verläufe der Kurven für Perzep-

tron 4 und 5 bereits deutlich größere Fehler aufweisen als die von Perzeptron 3. Dies ist darauf zurückzuführen, dass die Bestimmung der Muster schwieriger wird, je weiter in die Vergangenheit geschaut werden muss. Zwischen Ablesen des Zustands und Einspeisung des zu bestimmenden Musters ist das Liquid einer größeren Anzahl von Einflüssen ausgesetzt, die der Klassifizierer in seine Generalisierung aufnehmen muss.

Um den schwindenden Speicher des Liquids im Rahmen systematischerer Untersuchungen genauer zu charakterisieren, sollte ein Maß für die *Speicherkapazität* definiert werden (siehe [5, 3]). Zu diesem Zweck könnte beispielsweise für die einzelnen zu bestimmenden Muster eine Grenze für die Erkennungsrate bestimmt werden, ab der keine signifikante Korrektheit mehr zu erkennen ist. Die Zeit des letzten erkannten Musters würde dann das Maß der Speicherkapazität darstellen. Alternativ würde sich anbieten, beispielsweise eine Exponentialkurve an die Werte der Erkennungsraten in Abhängigkeit ihrer Musterzeiten anzupassen, um die Halbwertszeit dieser Kurve als Maß zu verwenden. Da diese Definitionen jedoch noch vom Erinnerungsvermögen τ des Klassifizierers abhängen, muss diesbezüglich eine Bereinigung stattfinden. Dafür könnte ein entsprechender Wert auch für den Klassifizierer bei direkter Anwendung auf den Input bestimmt werden, um einen Referenzwert zu erhalten. Eine Angabe relativ dazu würde ein aussagekräftiges Maß für die Speicherkapazität des Liquids darstellen.

4 Zusammenfassung und Ausblick

Im Rahmen der vorliegenden Arbeit wurde zum ersten Mal das sogenannte Liquid Computing auf einer spike-basierten, hoch konfigurierbaren und massiv beschleunigten neuromorphen Hardware umgesetzt. Es wurden experimentelle Daten präsentiert (siehe Kapitel 3.6), welche die Funktionsfähigkeit der auf der FACETS-Hardware implementierten Liquid State Machine belegen. Des weiteren wurde nachgewiesen, dass die gewählte Klassifizierungsaufgabe nicht trivial lösbar war: Ein Vergleich zwischen der Anwendung des Klassifizierers auf den unprozessierten Input und seiner Anwendung auf den Zustand des auf der FACETS-Hardware realisierten Liquids zeigte eindeutig dessen Speicherfähigkeit (siehe Abschnitt 3.6.2).

Selektive Parameterstudien hatten zum Ergebnis, dass sich bestimmte Parameterbereiche hinsichtlich der Klassifizierungsleistung besonders auszeichnen (siehe Abschnitt 3.6.1). Dabei stellte sich eine hohe Sensibilität der verwendeten Netzwerke bezüglich ihrer Konnektivität heraus. Es wurden zwei charakteristische Verhaltensweisen der Netzwerke beobachtet, wobei Experimente teilweise sehr hoher Aktivität an Bandbreitenprobleme der Kommunikation mit der Hardware stießen (siehe Abbildung 13). Der Übergang zwischen diesen durch Variation der Konnektivität zugänglichen Dynamikregimes wies die höchste Klassifizierungsleistung auf. Dies deckt sich mit den in [5, 42] präsentierten Ergebnissen der Perzeptron-Umsetzung des Liquids.

Zusätzlich wurden im Rahmen der Funktionalitätssicherung der Ansteuerungssoftware Beiträge geleistet. Es wurden für das im Abbildungsprozess verwendete Graphenmodell Komponententests entwickelt sowie Kalibrierungen, Analog- und Digitaltests an der FACETS-Hardware durchgeführt (siehe Kapitel 2.2).

Die vorgestellte Arbeit zeigt die Eignung der FACETS-Hardware für Liquid Computing und bildet damit nur das erste Glied in einer Reihe möglicher, aufeinander aufbauender Untersuchungen. Liquid Computing als Modell der Informationsverarbeitung ist bereits umfassend studiert [23, 24, 26, 25, 27, 28, 19]. Daher ist eine Fokussierung auf hardwarespezifische Fragestellungen notwendig, welche die Optimierung der Umsetzung von LSM auf neuromorphen Systemen adressieren. So sollte z.B. der nächste Schritt darin bestehen, den Klassifizierer in Hardware zu implementieren. Diesbezüglich entwickelt Sebastian Jeltsch im Rahmen seiner Diplomarbeit am Kirchhoff-Institut einen auf dem sogenannten *Tempotron* basierenden Klassifizierer, der die Funktionsweise der auf der Hardware modellierten IF-Neuronen zugrunde legt und sich daher für eine reine Hardware-Lösung eignet.

Des weiteren lässt der im Rahmen dieser Arbeit beobachtete schmale Bereich zwischen zu niedriger und zu hoher Netzwerkaktivität die Vermutung zu, dass die Verwendung von dynamischen Synapsen (STP) eine Verbesserung bewirken kann: Die in Abschnitt 3.5 beobachteten spontanen Übergänge zwischen den Dynamikregimes könnten so möglicherweise verhindert werden und sich insgesamt ein größerer dynamischer Bereich eröffnen. Auch ergeben die Simulationen in [24], dass STP das Klassifizierungsvermögen der LSM deutlich erhöht.

Ferner müsste untersucht werden, wie robust sich die implementierten Netzwerke bezüglich der Verwendung auf unterschiedlichen Spikey-Chips verhalten. Außerdem spielt bei der Parameteroptimierung möglicherweise auch der Typ der Klassifizierungsaufgabe eine Rolle. Die systematische Untersuchung dieser Abhängigkeiten ist entscheidend für die Frage, ob die Angabe allgemein gültiger Liquidparameter möglich ist.

Für einen späteren Zeitpunkt sollte in Betracht gezogen werden, Liquid Computing auf die FACETS Stage 2 Hardware abzubilden. Dabei stellt sich die Frage, wie sich die Skalierung des Liquids hin zu deutlich größeren Netzwerken auf seine Klassifizierungsfähigkeit auswirkt. Die Simulationen von Maass et al. in [24] zeigen, dass eine modulare Vergrößerung des Liquids durch voneinander isolierte Netzwerke die Separationseigenschaft und Erkennungsrate erhöht. Eine solche Konfiguration würde sich für die FACETS Stage 2 Hardware sehr gut eignen: Die von ihr

zur Verfügung gestellten Ressourcen könnten auf diese Weise optimal ausgeschöpft werden, ohne dabei auf Verbindungsengpässe zu stoßen.

A Anhang

A.1 Quellcode

Der Quellcode für die im Rahmen dieser Arbeit erstellten Unit Tests sind im git-Repository zu finden unter:

```
git@gitviz.kip.uni-heidelberg.de:symap2ic.git
```

Das entsprechende Unterverzeichnis ist:

```
symap2ic/components/pynnhw/test/unittests/c++/
```

Die Verwaltung bzw. Kommunikation von Fehlerberichten geschieht über das Indefero-System mit Hilfe sogenannter *Issues*, abrufbar unter der Webseite des symap2ic-Projekts:

```
https://gitviz.kip.uni-heidelberg.de/index.php/p/symap2ic/
```

Die aus den Unit Tests resultierenden Fehlerberichte befinden sich unter:

```
https://gitviz.kip.uni-heidelberg.de/index.php/p/symap2ic/issues/
```

Die Kalibrierungs- und Testergebnisse der Spikey-Chips sind in das Wiki der Electronic Vision(s) Group eingetragen:

```
https://wiki.kip.uni-heidelberg.de/KIPwiki/index.php/Visions_Privat:Spikey4LabLog
```

Folgendes Repository enthält die PyNN-Umsetzung der Liquid State Machine:

```
https://gitviz.kip.uni-heidelberg.de/index.php/p/PyNN-Scripts/
```

A.2 Liquid-Parameter

Tabelle 1 enthält die für die Parameter aus Abschnitt 3.1 verwendeten Werte. Des weiteren wurden die Experimente auf Backplane 1 und Spikey 444 mit den entsprechenden Kalibrationsdateien verwendet.

Beschreibung	**Parameter**	**Wert**
Anzahl Neurone	N	135
Dimensionen	$n_\mathrm{x} \times n_\mathrm{y} \times n_\mathrm{z}$	$3 \times 3 \times 15$
Verhältnis inh./exz. Neurone	ratio_{inh}	$\frac{1}{5}$
Membrankapazität	C_m	$0,2\,\mathrm{nF}$
Ruheleitfähigkeit	g_leak	$6,7\,\mathrm{nS}$
Ruhepotenzial	V_rest	$-60\,\mathrm{mV}$
Feuerschwelle	V_thresh	$-55\,\mathrm{mV}$
Resetpotenzial	V_reset	$-65\,\mathrm{mV}$
Skalierungsfaktoren Verbindungswahrsch. (siehe Kap. 3.1)	$(C_\mathrm{EE}, C_\mathrm{EI}, C_\mathrm{II}, C_\mathrm{IE})$	$\left(\frac{1}{5}, \frac{1}{2}, \frac{1}{10}, \frac{2}{5}\right)$
Synaptisches Basisgewicht	g_max	$1\,\mathrm{nS}$
Skalierungsfaktoren synaptische Gewichte (siehe Kap. 3.1)	$(w_\mathrm{EE}, w_\mathrm{EI}, w_\mathrm{II}, w_\mathrm{IE})$	$(5, 12, 2, 12)$
Synaptische Zeitkonstante (exz. und inh.)	τ_syn	$5\,\mathrm{ms}$
Anzahl Stimulations-Spiketrains	N_input	40

Tabelle 1: Parameterwerte in ihrer biologischen Interpretation

Literatur

[1] D. Abrahams and R.W. Grosse-Kunstleve. Building hybrid systems with Boost.Python, 2003.

[2] Roland Baddeley, L. F. Abbott, Michael C. A. Booth, Frank Sengpiel, Toby Freeman, Edward A. Wakeman, and Edmund T. Rolls. Responses of neurons in primary and inferior temporal visual cortices to natural scenes. *Proceedings of the Royal Society B*, 264:1775–1783, 1997.

[3] N. Bertschinger and T. Natschläger. Real-time computation at the edge of chaos in recurrent neural networks. *Neural Computation*, 16(7):1413 – 1436, July 2004.

[4] R. Brette, M. Rudolph, T. Carnevale, M. Hines, D. Beeman, J. M. Bower, M. Diesmann, A. Morrison, P. H. Goodman, F. C. Harris Jr, M. Zirpe, T. Natschlager, D. Pecevski, B. Ermentrout, M. Djurfeldt, A. Lansner, O. Rochel, T. Vieville, E. Muller, A. P. Davison, S. El Boustani, and A. Destexhe. Simulation of networks of spiking neurons: A review of tools and strategies, 2006.

[5] Daniel Brüderle. Implementing spike-based computation on a hardware perceptron. Diploma thesis (English), University of Heidelberg, HD-KIP-04-16, 2004.

[6] Daniel Brüderle. *Neuroscientific Modeling with a Mixed-Signal VLSI Hardware System*. PhD thesis, 2009.

[7] R. Cossart, D. Aronov, and R. Yuste. Attractor dynamics of network up states in the neocortex. *Nature*, 423:238–283, 2003.

[8] A. P. Davison, D. Brüderle, J. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger. PyNN: a common interface for neuronal network simulators. *Front. Neuroinform.*, 2(11), 2008.

[9] Andrew Davison, Eilif Muller, Daniel Brüderle, and Jens Kremkow. A common language for neuronal networks in software and hardware. *The Neuromorphic Engineer*, 2010.

[10] Alain Destexhe. Conductance-based integrate-and-fire models. *Neural Comput.*, 9(3):503–514, 1997.

[11] Markus Diesmann and Marc-Oliver Gewaltig. NEST: An environment for neural systems simulations. In Theo Plesser and Volker Macho, editors, *Forschung und wisschenschaftliches Rechnen, Beiträge zum Heinz-Billing-Preis 2001*, volume 58 of *GWDG-Bericht*, pages 43–70. Ges. für Wiss. Datenverarbeitung, Göttingen, 2002.

[12] Rodney Douglas, Henry Markram, and Kevan Martin. *The Synaptic Organization in the Brain*, chapter Neocortex, pages 499–558. Oxford University Press, 5 edition, 2004.

[13] Wulfram Gerstner and Werner Kistler. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002.

[14] Andreas Grübl. *VLSI Implementation of a Spiking Neural Network*. PhD thesis, Ruprecht-Karls-University, Heidelberg, 2007. Document No. HD-KIP 07-10.

[15] Andrew Hunt and David Thomas. *Unit-Tests mit JUnit*. Number 2 in Pragmatisch Programmieren ; 2 ; Pragmatisch Programmieren. Hanser, München ; Wien, 2004.

[16] G. Indiveri, E. Chicca, and R. Douglas. A VLSI array of low-power spiking neurons and bistable synapses with spike-timing dependent plasticity. *IEEE Transactions on Neural Networks*, 17(1):211–221, Jan 2006.

[17] ISO/IEC 14882. *Programming Language C++*, July 1998.

[18] H. Jaeger. The "echo state" approach to analysing and training recurrent neural networks. Technical Report GMD Report 148, German National Research Center for Information Technology, 2001.

[19] Herbert Jaeger, Wolfgang Maass, and Jose Principe. Special issue on echo state networks and liquid state machines. *Neural Networks*, 20(3):287–289, April 2007.

[20] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice-Hall software series. Prentice-Hall, Englewood Cliffs, N.J., 2. ed. edition, 1988.

[21] S. B. Kotsiantis. Supervised machine learning: A review of classification techniques. *Informatica*, 31:249–268, 2007.

[22] C. G. Langton. Computation at the edge of chaos. *Physica D*, 42, 1990.

[23] W. Maass. Networks of spiking neurons: the third generation of neural network models. *Neural Networks*, 10:1659–1671, 1997.

[24] W. Maass, T. Natschläger, and H. Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, 14(11):2531–2560, 2002.

[25] W. Maass, T. Natschläger, and H. Markram. *Computational models for generic cortical microcircuits*, chapter 18, pages 575–605. Number ISBN 1-58488-362-6. J. Feng, Boca Raton, 2004.

[26] W. Maass, T. Natschlager, and H. Markram. Fading memory and kernel properties of generic cortical microcircuit models. *J Physiol Paris*, 98(4-6):315–30, 2004. Institute for Theoretical Computer Science, Technische Universitaet Graz, Austria. maass@igi.tugraz.at.

[27] W. Maass, T. Natschläger, and H. Markram. On the computational power of circuits of spiking neurons. *Journal of Physiology (Paris)*, (in press), 2004.

[28] Wolfgang Maass, Prashant Joshi, and Eduardo D. Sontag. Computational aspects of feedback in neural circuits. *PLoS Computational Biology*, 3(1):e165+, January 2007.

[29] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, pages 127–147, 1943.

[30] C. A. Mead. *Analog VLSI and Neural Systems*. Addison Wesley, Reading, MA, 1989.

[31] Bertrand Meyer. *Object oriented software construction*. Prentice Hall international series in computer science. Prentice-Hall, New York [u.a.], 10. [pr.] edition, 1992.

[32] Morrison, Abigail, Diesmann, Markus, Gerstner, and Wulfram. Phenomenological models of synaptic plasticity based on spike timing. *Biological Cybernetics*, 98(6):459–478, June 2008.

[33] Abigail Morrison, Carsten Mehring, Theo Geisel, Ad Aertsen, and Markus Diesmann. Advancing the boundaries of high connectivity network simulation with distributed computing. *Neural Comput.*, 17(8):1776–1801, 2005.

[34] Eric Müller. Operation of an imperfect neuromorphic hardware device. Diploma thesis (English), University of Heidelberg, HD-KIP-08-43, 2008.

[35] T. Natschläger, W. Maass, and H. Markram. The "liquid computer": A novel strategy for real-time computing on time series. *Special Issue on Foundations of Information Processing of TELEMATIK*, 8(1):39–43, 2002.

[36] T. Natschläger, H. Markram, and W. Maass. Computer models and analysis tools for neural microcircuits. In R. Kötter, editor, *A Practical Guide to Neuroscience Databases and Associated Tools*, chapter 9. Kluver Academic Publishers (Boston), 2002.

[37] Guido Van Rossum. *Python Reference Manual: February 19, 1999, Release 1.5.2*. iUniverse, Incorporated, 2000.

[38] J. Schemmel, D. Brüderle, K. Meier, and B. Ostendorf. Modeling synaptic plasticity within networks of highly accelerated I&F neurons. In *Proceedings of the 2007 IEEE International Symposium on Circuits and Systems (ISCAS'07)*. IEEE Press, 2007.

[39] J. Schemmel, J. Fieres, and K. Meier. Wafer-scale integration of analog neural networks. In *Proceedings of the 2008 International Joint Conference on Neural Networks (IJCNN)*, 2008.

[40] J. Schemmel, A. Grübl, K. Meier, and E. Muller. Implementing synaptic plasticity in a VLSI spiking neural network model. In *Proceedings of the 2006 International Joint Conference on Neural Networks (IJCNN'06)*. IEEE Press, 2006.

[41] J. Schemmel, S. Hohmann, K. Meier, and F. Schürmann. A mixed-mode analog neural network using current-steering synapses. *Analog Integrated Circuits and Signal Processing*, 38(2-3):233–244, 2004.

[42] F. Schürmann, K. Meier, and J. Schemmel. Edge of Chaos Computation in Mixed Mode VLSI – "A Hard Liquid". In L.K. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems 17*, Cambride, 2004. MIT Press.

[43] Gordon M. [Hrsg.] Shepherd, editor. *The synaptic organization of the brain*. Oxford Univ. Press, Oxford [u.a.], 5. ed. edition, 2004.

[44] Georg Erwin Thaller. *Software-Test*. Heise, Hannover, 2., aktualisierte u. erw. aufl. edition, 2002.

[45] Anita M. Turner and William T. Greenough. Differential rearing effects on rat visual cortex synapses. i: Synaptic and neuronal density and synapses per neuron. *Brain Research*, 329:195–203, 1985.

[46] K. Wendt, M. Ehrlich, and R. Schüffny. Gmpath - a path language for navigation, information query and modification of data graphs. In *Proceedings of the Artificial Neural Networks and Intelligent Information Processing Conference (ANNIIP) 2010*, pages 31–42, 2010.

[47] Karsten Wendt, Matthias Ehrlich, and René Schüffny. A graph theoretical approach for a multistep mapping software for the facets project. In *CEA'08: Proceedings of the 2nd WSEAS International Conference on Computer Engineering and Applications*, pages 189–194, Stevens Point, Wisconsin, USA, 2008. World Scientific and Engineering Academy and Society (WSEAS).

Erklärung

Ich versichere, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, 6. August 2010

.....
(signature)