



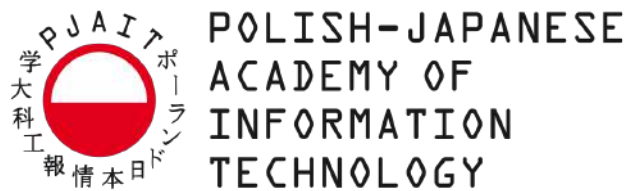
Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



Neural Simulation Pipeline for Liquid State Machines

Neuroinformatics, Scientific Workflows, Containerisation,
Computer Simulations, Liquid State Machines



Karol Chlasta

Supervisor: dr hab. Grzegorz M. Wójcik, prof. PJAiT

Advisor: dr hab. Izabela Krejtz, prof. USWPS

Department of Computer Science
Polish-Japanese Academy of Information Technology

This dissertation is submitted for the degree of

Doctor of Philosophy

ICT & Psychology

March 2023

I would like to dedicate this thesis to my loving wife Anna.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This PhD dissertation contains 45,534 words including appendices, bibliography, footnotes, tables and equations. It has 62 figures, 36 tables and 247 references.

Karol Chlasta
March 2023

Acknowledgements

I am grateful to Grzegorz Marcin Wójcik and Izabela Krejtz, who have provided me with a constant stream of ideas, encouragement and feedback throughout my studies at ICT & Psychology Doctoral Programme. I would also like to thank Krzysztof Krejtz (SWPS University), who sparked my interest in visual system and eye-tracking; to Paweł Sochaczewski (Aviva) for his help with Neural Simulation Pipeline's scripts, Aneta Sobotka (Technicus) for her feedback as an independent software tester, Szymon Chlasta (TechnipFMC) for his help with modification of the Neural Simulation Cluster's top cover, as well as Dariusz Jemielniak (Kozminski University), Paweł Holas (University of Warsaw), Andreas Polze and Tobias Pape (Hasso Plattner Institute, University of Potsdam) for all the interesting discussions.

Next, I am indebted to Leanne Bucknall (Technicus) and Michael Connolly (Aviva) for the proofreading of the text, and all the encouragement to progress with the preparation of this thesis.

I am grateful to Polish-Japanese Academy of Information Technology and Polish National Centre for Research and Development for partially funding my research by granting me a PhD scholarships while being an ICT & Psychology student. I am also grateful to Future SOC Lab at Hasso Plattner Institute (University of Potsdam) for granting me access to their data centre, so that I could run my neural simulations.

Finally, I would like to thank my wife, Anna, for her love and support, and for sharing her life with me.

Abstract

English language version

Author claims that numerical simulations must integrate a robust model development methodology, with adequate testing and simulation steering workflows to increase scientific throughput and improve utilisation of current and next-generation computational infrastructure, available both on-premise and in-cloud. To this end, there is the need to transform the end-to-end computational experiment workflow from one that is non-universal and manual to one that is standardised and (at least partially) automated.

Liquid State Machines (LSMs) are a type of recurrent neural network that have been widely used for tasks such as pattern recognition and classification. However, simulating LSMs can be computationally expensive due to their large number of neurons and connections. In this PhD thesis, author presents a novel Neural Simulation Pipeline (NSP) for LSMs that significantly reduces the computational cost of simulation while automating the tasks needed to manage and deploy the experiments into different execution environments. A provider-agnostic simulation framework can be used to run simulations on different hardware platforms or microprocessor architectures to allow researchers to use the most appropriate hardware and software for their specific simulation needs, without being tied to a specific vendor or provider. In the High Performance Computing (HPC) context, public cloud resources are becoming an alternative to the expensive on-premise clusters.

This thesis presents Neural Simulation Pipeline (NSP), a set of Bash and PowerShell scripts to facilitate the large scale computer simulations and their deployment to multiple computer infrastructures using the infrastructure as code (IaC) containerisation approach. The pipeline consists of three main components: a data preprocessing module, a simulation module, and a post-processing module. The preprocessing module manages the experiment's input data into a format suitable for LSM simulation, while the simulation module performs the actual experiment execution using a selected simulation engine. The post-processing module then analyses the simulated data and generates the final results.

Author demonstrates the effectiveness of NSP in a pattern recognition task programmed with GENESIS (a general purpose simulation engine for neural systems) and simulated through two custom-built visual systems: (1) RetNet(4x8,1) based on a single LSM column of multiple sizes, and (2) RetNet(28x28,4) using four LSM columns. Both systems were built using biologically plausible Hodgkin–Huxley spiking neurons. and are explored in the experimental chapters. The key finding relates to twelve different LSM readout algorithms, evaluated through five standard classification metrics, using the 10-fold cross-validation process. The LSM system presented achieves a repeatable accuracy and F1 Score of 81% for the readout based on Light Gradient Boosting Machine.

Moreover, the pipeline is evaluated by performing additional 54 experiments executed on-premise, and in the AWS Public Cloud environment. Author compares the standard and containerised execution, as well as presents the cost of execution in AWS. The results show that the NSP can significantly reduce entry barriers to LSM simulations, making it more practical and cost effective for real-world applications. The experimental conclusions are supplemented with practical tips related to prototyping with author's custom single board computer cluster (Neural Simulation Cluster), and suggested further research on the pipeline.

Polish language version

Autor twierdzi, że symulacje numeryczne powinny integrować metodologiczne podejście do rozwoju i testowania modeli cybernetycznych, z odpowiednimi przepływami pracy obliczeniowej. Uczynić to należy w celu zwiększenia ich skuteczności naukowej oraz poprawy wykorzystania obecnej i przyszłej infrastruktury obliczeniowej, dostępnej zarówno w modelu tradycyjnym, jak i w chmurowym środowisku obliczeniowym. W tym celu istnieje potrzeba przekształcenia przepływu pracy dla eksperymentów obliczeniowych z takiego, który jest nieuniwersalny i manualny na taki, który jest znormalizowany i (przynajmniej częściowo) zautomatyzowany.

Maszyny stanów płynowych (ang. Liquid State Machines, lub w skrócie LSM) są rodzajem rekurencyjnej sieci neuronowej, która jest szeroko stosowana w zadaniach takich jak rozpoznawanie i klasyfikacja wzorców. W niniejszej pracy autor przybliży tę tematykę i wskazuje, że symulacje takie mogą być kosztowne obliczeniowo ze względu na dużą liczbę wiarygodnych, kolczastych neuronów pulsacyjnych Hodgkina-Huxleya i mnogość połączeń między nimi. W niniejszej pracy doktorskiej autor przedstawia nowatorski Potok Symulacji Neuronowej (ang. Neural Simulation Pipeline, lub w skrócie NSP) dla maszyn LSM, który znacząco zmniejsza koszt obliczeniowy takich symulacji, automatyzując jednocześnie

zadania potrzebne do zarządzania i wdrażania eksperymentów obliczeniowych w różnych środowiskach uruchomieniowych. Potok umożliwia przeprowadzanie symulacji komputerowych w standaryzowany sposób na różnych platformach programowych i sprzętowych (w tym architekturach mikroprocesorowych), ułatwiając badaczom wykorzystanie najbardziej odpowiedniego sprzętu i oprogramowania dla ich specyficznych potrzeb symulacyjnych, bez przywiązywania ich do konkretnego dostawcy usług lub architektury. W kontekście super-obliczeń HPC (ang. High Performance Computing), zasoby chmury publicznej stają się alternatywą dla drogich klastrów komputerowych działających w tradycyjnych centrach obliczeniowych.

Wspomniany potok symulacji neuronowej NSP, będący zestawem skryptów Bash i PowerShell ułatwiających przeprowadzanie symulacji komputerowych na większą skalę i wdrażanie ich w różnych środowiskach uruchomieniowych korzysta z podejścia kontenerowego w paradygmacie Infrastruktura jako kod (ang. Infrastructure as Code, lub w skrócie IaC). Sam potok składa się z trzech głównych komponentów logicznych: modułu wstępnego przetwarzania danych, modułu symulacyjnego oraz modułu powykonaniowego. Moduł przetwarzania wstępnego zarządza danymi wejściowymi eksperymentu do formatu odpowiedniego dla symulacji LSM, podczas gdy moduł symulacji wykonuje rzeczywiste wykonanie eksperymentu przy użyciu wybranego silnika symulacyjnego. Moduł powykonaniowy wspomaga analizę danych symulacyjnych i generuje ostateczne wyniki.

Autor demonstruje skuteczność NSP w zadaniu rozpoznawania wzorców zaprogramowanym w GENESIS (silnik symulacyjny ogólnego przeznaczenia dla systemów neuronowych) i symulowanym przez dwa zbudowane przez siebie systemy wizyjne: (1) RetNet(8x5,1) oparty na pojedynczej kolumnie LSM o zmiennym rozmiarze oraz (2) RetNet(28x28,4) wykorzystującym cztery kolumny LSM. Oba systemy zostały zbudowane z wykorzystaniem biologicznie wiarygodnych neuronów impulsowych Hodgkina-Huxleya i są omówione w rozdziałach eksperymentalnych niniejszej rozprawy. Kluczowy wynik badawczy dla zaproponowanej architektury maszyn stanów płynowych dotyczy zbadania dwunastu różnych algorytmów warstwy odczytującej, ocenionych za pomocą pięciu standardowych metryk oceny klasyfikacji, z wykorzystaniem procesu 10-krotnej walidacji krzyżowej. Uzyskany przez autora wynik zaproponowanego systemu LSM osiąga dokładność (oraz F1 Score) na poziomie 81%, dla odczytu opartego o algorytm drzew decyzyjnych LightGBM (ang. Light Gradient Boosting Machine).

Ponadto, potok jest oceniany poprzez wykonanie 54 dodatkowych eksperymentów obliczeniowych wykonywanych w tradycyjnym centrum danych i w środowisku publicznej chmury obliczeniowej AWS. Autor porównuje standardowe i skonteneryzowane wykonania symulacji maszyn stanów płynowych, jak również przedstawia koszt wykonania poszczegól-

nych eksperymentów w AWS. Wyniki pokazują, że zaproponowane rozwiązanie może znacząco zmniejszyć bariery wejścia do prowadzenia takich symulacji, czyniąc je bardziej praktycznymi w zastosowaniach i efektywnymi kosztowo. Wnioski z eksperymentów są uzupełnione o praktyczne wskazówki związane z prototypowaniem przy użyciu autorskiego klastra obliczeniowego opartego o komputery jednopłytkowe (Neural Simulation Cluster), oraz sugestie dotyczące dalszych prac badawczych i wdrożeniowych dla potoku symulacji neuronowej NSP.

Table of contents

List of figures	xvii
List of tables	xxi
Nomenclature	xxiii
1 Modelling Networks of Spiking Neurons	1
1.1 Introduction	1
1.2 Problem, Motivation, and Proposed Solution	2
1.3 Simulating Neural Networks on HPC Resources	3
1.4 Integrating Pipeline into Neural Simulations	5
1.5 Limitations of the Current State of the Art (EEG)	6
1.6 Visual Cortex	6
1.6.1 Visual Cortex as Part of Cerebral Cortex	6
1.6.2 Hypercolumns as Information Processing Units	9
1.7 Simulating Spiking Neural Networks	11
1.7.1 Hodgkin-Huxley Model	11
1.7.2 Leaky Integrate-and-Fire Model	12
1.7.3 Izhikevich Model	13
1.8 Challenges and Progress	14
1.9 Current State of the Art in Simulation Frameworks	15
1.10 Simulation System	16
1.10.1 General Neural Simulation System	16
1.10.2 Other Simulation Frameworks	22
1.11 Thesis Statement	23
2 Liquid Computing in Brain Modelling	27
2.1 Introduction	27
2.2 From Hardware and Software to Brain and Mind	28

2.3	Liquid State Machine Concept	30
2.4	Structural Components of Liquid State Machines	31
2.5	Liquid Computations with Liquid State Machines	32
2.6	Mathematical description of Liquid State Machines	34
2.7	Discussion of Selected Properties of Liquid State Machines	35
2.7.1	Separation Property	36
2.7.2	Approximation Property	38
2.8	The Role of Readout Layer	38
2.9	State of Practice	40
2.9.1	The Concept of Echo State Networks	41
2.9.2	Extreme Learning Machines	43
2.10	Large Scale Computer Simulations of Liquid Models	46
2.11	Summary	50
3	Liquid State Machine Models	53
3.1	Introduction	53
3.2	Modelling of Biological Systems	54
3.3	Brain Simulations and Order of Growth	56
3.4	RetNet(28x28,4)	59
3.5	Wide RetNet(8x5,1)	62
3.6	Deep RetNet(8x5,1)	63
3.7	Simulation Setting	64
3.7.1	Initial Setup at Google Colab	64
3.7.2	Raspberry Pi and ROCKPro64 Single Board Computers	65
3.7.3	HPI Future SOC Lab HPC Resources	68
3.7.4	AWS Cloud Computing Services	68
4	Experimental Work	73
4.1	Introduction	73
4.2	Experiments with RetNet(28x28,4)	74
4.3	Computational Complexity of Single LSM Column	77
4.4	Experiments with RetNet(8x5,1)	81
4.4.1	Wide RetNet(8x5,1)	81
4.4.2	Deep RetNet(8x5,1)	82
4.5	Evaluating Liquid State Machine in RetNet(8x5,1)	86
4.5.1	Constructing LSM Readout Process	86
4.5.2	Machine Learning Algorithms	91

4.5.3	Experiment Metrics	91
4.5.4	Detailed Results for LightGBM	94
4.5.5	Detailed Results for Gradient Boosted Trees	96
4.5.6	Detailed Results for XGBoost	96
4.5.7	Detailed Results for Extra Trees	98
4.5.8	Detailed Results for Random Forest	100
4.5.9	Detailed Results for Logistic Regression Max Entropy	102
4.5.10	Detailed Results for Logistic Regression Stochastic Gradient Descent	104
4.5.11	Detailed Results for Multi-layer Perceptron	106
4.5.12	Detailed Results for LASSO-LARS	108
4.5.13	Detailed Results for Support Vector Machines	113
4.5.14	Detailed Results for Decision Tree	115
4.5.15	Detailed Results for AdaBoost Classifier	117
4.6	Experimental Summary	119
5	Neural Simulation Pipeline	121
5.1	Introduction	121
5.2	Prototyping with Single Board Computer Clusters	122
5.3	Neural Simulation Cluster	125
5.4	Neural Simulations Pipeline	130
5.4.1	Components	130
5.4.2	Architecture	131
5.4.3	Containerisation with Docker	147
5.4.4	Docker Architecture	151
5.4.5	Docker Kernel Internals	152
5.4.6	Docker Engine and NSP Scripts	155
5.4.7	Neural Simulation Pipeline Container Scripts	155
5.5	Neural Simulation Pipeline Experimental Evaluation	160
5.6	Neural Simulation Pipeline Limitations	163
6	Summary and Conclusion	165
6.1	Summary	165
6.2	Future Directions	166
6.3	Practical Significance	168
6.4	Key Insights	169
6.5	Conclusion	170

References	173
Appendix A Academic & Professional Profile	191
A.1 Karol Chlasta’s Resumé	191
A.2 Selected Formal Education	192
A.3 Professional Experience in IT Sector	192
A.4 Selected IT Certificates	192
A.5 Academic Experience & Achievements	193
Appendix B Building Neural Simulation Cluster	197
B.1 Bill of Materials for Neural Simulation Cluster	197
B.1.1 Assembling Hardware Elements	200
B.2 Configuring Neural Simulations Cluster	207
B.2.1 Cluster Configuration Steps	213
Appendix C Additional Information on Neuronal Models	215
Appendix D Neural Simulation Pipeline	219
D.1 Repositories	219
D.2 Usage	219
D.2.1 Neural Simulation Pipeline Installation Steps	219
D.2.2 Neural Simulation Pipeline Simulation Execution Steps	221
D.2.3 Neural Simulation Pipeline Finalisation and Cleanup Steps	226
D.3 Scripts	227

List of figures

1.1	Visual system in humans	8
1.2	Icecube model of a cat V1 hypercolumn	10
1.3	The equivalent circuit for a standard GENESIS neural compartment	17
1.4	A Purkinje cerebellar cell built with GENESIS	20
1.5	GENESIS' high-level architecture	21
1.6	Diagram of Liquid State Machine	24
2.1	Diagram of Liquid State Machine	32
2.2	The separation property of LSM in Euclidean and Bray-Curtis measures.	37
2.3	Application of LSM model to the speech recognition	39
2.4	Jaeger's basic ESN architecture	42
2.5	IBM BlueGene supercomputers used by Blue Brain Project.	47
2.6	Evolution of the Blue Brain Project into Human Brain Project	49
3.1	Retina of Model RetNet(28x28,4)	60
3.2	Structure of LSM column in RetNet(28x28,4)	61
3.3	Retina of Model RetNet(5x8,1)	62
3.4	High level simulation setup including GENESIS and Spark	66
3.5	A view of NSC's Raspberry Pi 4 Model B node	69
3.6	300 ms of simulation of two Hodgkin-Huxley neurons on two NSC nodes	70
3.7	NSP Architecture Diagram	72
4.1	Frequency of retina spikes in the 1st sec of simulation in RetNet(28x28,4)	75
4.2	Cumulative retina spikes in the 1st sec of simulation in RetNet(28x28,4)	76
4.3	Visual signal processing in the 1st and 2nd LSM column of the RetNet(28x28,4)	78
4.4	Visual signal processing in the 3rd and 4th LSM column of RetNet(28x28,4)	79
4.5	Computational complexity of RetNet(8x5,1)	80
4.6	Spiking activity for RetNet(8x5,1) growing from 84 to 12044 HH neurons	81
4.7	RetNet(8x5,1) simulation time on HPI vs own NSC (RPi4B)	83

4.8	RetNet(8x5,1) simulation time for different input patterns on HPI	84
4.9	RetNet(8x5,1) simulation time for different thread types on HPI	85
4.10	Model selection in LSM readout prototyping	90
4.11	Additional LSM readout details for LightGBM algorithm	95
4.12	Additional LSM readout details for Gradient Boosted Trees algorithm	97
4.13	Additional LSM readout details for XGBoost algorithm	99
4.14	Additional LSM readout details for Extra Trees algorithm	101
4.15	Additional LSM readout details for Random Forest algorithm	103
4.16	Additional LSM readout details for Logistic Regression algorithm	105
4.17	Additional LSM readout details for Stochastic Gradient Descent algorithm	107
4.18	Additional LSM readout details for Multi-layer Perceptron algorithm	110
4.19	Additional LSM readout details for LASSO-LARS algorithm	112
4.20	Additional LSM readout details for Support Vector Machine algorithm	114
4.21	Additional LSM readout details for Decision Tree algorithm	116
4.22	Additional LSM readout details for AdaBoost algorithm	118
5.1	Exemplary SBC Clusters.	124
5.2	Neural Simulations Cluster Design Diagram	126
5.3	Photos of assembled Neural Simulation Cluster.	127
5.4	Technical drawing of Neural Simulation Cluster's top cover modification	128
5.5	Schematic diagram of Neural Simulation Pipeline	132
5.6	User's workflow in Neural Simulation Pipeline	132
5.7	Developer's workflow in Neural Simulation Pipeline	133
5.8	High-level view of Neural Simulation Pipeline	137
5.9	Detailed experiment workflow in Neural Simulation Pipeline	148
5.10	Docker Architecture	153
5.11	Evaluating NSP: Average CPU time for each RetNet size and execution type	161
5.12	Evaluating NSP: Average memory for each RetNet size and execution type	162
5.13	Evaluating NSP: Cost structure in AWS	162
B.1	NSC's three RPr SBCs screwed together using the standoffs	201
B.2	NSC's power distribution unit and power cables	202
B.3	NSC's power supply and its wiring	203
B.4	NSC's Gigabit Ethernet switch	204
B.5	NSC's internal cabling	205
B.6	NSC's front panel assembled	206
B.7	NSC up and running	208

C.1 Retina of RetNet(8x5,1) with a synaptic stimulus 217

List of tables

4.1	Euclidean distance between 4 LSM columns in RetNet(28x28,4)	77
4.2	Summary of RetNet(8x5,1) and RetNet(28x28,4) simulations (HPI vs NSC)	82
4.3	Neural Simulation Pipeline Execution (Bare-bone HPI)	87
4.4	Neural Simulation Pipeline Execution (Containerised HPI)	88
4.5	Neural Simulation Pipeline Execution (Containerised AWS)	89
4.6	Summary of performance for different LSM readouts in RetNet(5x8,1) . . .	93
4.7	Details of LSM readout using the LightGBM algorithm	94
4.8	Confusion matrices for LSM readout using LightGBM algorithm	94
4.9	Details of LSM readout using the Gradient Boosted Trees algorithm	96
4.10	Confusion matrices for LSM readout using Gradient Boosted Trees algorithm	96
4.11	Details of LSM readout using the XGBoost algorithm	98
4.12	Confusion matrices for LSM readout using XGBoost algorithm	100
4.13	Details of LSM readout using the Extra Trees algorithm	100
4.14	Confusion matrices for LSM readout using Extra Trees algorithm	102
4.15	Details of LSM readout using the Random Forest algorithm	102
4.16	Confusion matrices for LSM readout using Random Forest algorithm . . .	104
4.17	Details of LSM readout using the Logistic Regression algorithm	104
4.18	Confusion matrices for LSM readout using Logistic Regression algorithm .	106
4.19	Details of LSM readout using the SGD algorithm	106
4.20	Confusion matrices for LSM readout using SGD algorithm	108
4.21	Details of LSM readout using the Multi-layer Perceptron	109
4.22	Confusion matrices for LSM readout using Multi-layer Perceptron	109
4.23	Details of LSM readout using the LASSO-LARS algorithm	111
4.24	Confusion matrices for LSM readout using LASSO-LARS algorithm	111
4.25	Details of LSM readout using the Support Vector Machine algorithm	113
4.26	Confusion matrices for LSM readout using Support Vector Machine algorithm	113
4.27	Details of LSM readout using the Decision Tree algorithm	115

4.28	Confusion matrices for LSM readout using Decision Tree algorithm	116
4.29	Details of LSM readout using the AdaBoost algorithm	117
4.30	Confusion matrices for LSM readout using AdaBoost algorithm	117
5.1	List of user and container scripts in Neural Simulation Pipeline	136
A.1	Author's selected peer-reviewed publications	195
B.1	Hardware specification of Raspberry Pi 4 Model B	198
B.2	Hardware specification of ROCKPro64	198
C.1	RetNet standard simulation parameters for HHLSMs	215
D.1	Scripts in Neural Simulation Pipeline and in Neural Simulation Cluster . . .	228

Nomenclature

Roman Symbols

mA milliamperere measures electrical current and is equal to one-thousandth of an ampere, $10^{-3} A = 0.001 A$

mV millivolt measures electric potential or electromotive force and is equal of one thousandth of a volt, $10^{-3} V = 0.001 V$

V volt, SI base unit of derived unit for electric potential and its difference (voltage)

Greek Symbols

α parameter indicating level 1 regularisation, a basic way to avoid overfitting in Machine Learning by penalising high-valued coefficients

γ parameter indicating minimal loss reduction to split a leaf for Machine Learning tree-based algorithms

λ parameter indicating level 2 regularisation in Machine Learning, implying a trade-off between more bias with low lambda, and less bias with high lambda (higher variance)

Ω the ohm, SI derived unit of electrical resistance

Subscripts

$\alpha_m, \alpha_n, \alpha_h, \beta_m, \beta_n, \beta_h$ variables describing a stochastic dynamics of the activation process with rate constants α and β in the HH neuron model

C_m membrane capacitance

E_k equilibrium potential (or reversal potential of the equivalent circuit)

E_l potential reducing the net channel current to zero when $V_m = E_r$

E_m	associated equilibrium potential (typically close to the E_r)
E_r	rest potential (of the equivalent circuit)
G_l	leakage conductance ($G_l = \frac{1}{R_m}$)
I_i	current source (optional injection of current)
$R_a R'_a$	axial resistances (on the sides of the equivalent circuit)
R_m	membrane resistance (of the equivalent circuit)
V_m	membrane potential, the potential inside a cell compartment (in relation to a point outside of the cell)
V_m	membrane potential, the potential inside a cell compartment (in relation to a point outside of the cell)

Acronyms / Abbreviations

<i>A/C</i>	Alternating current power, often provided to a device through a power supply
<i>ANN</i>	Artificial Neural Network (traditional; computes all neurons, fixed number of synapses)
<i>AP</i>	Approximation Property of LSM
<i>API</i>	Application Programming Interface
<i>ARM64</i>	Advanced RISC Machines, originally Acorn RISC Machine (64-bit extension)
<i>AUC</i>	Area under Curve (the ROC curve)
<i>BBP</i>	Blue Brain Project
<i>BRAIN</i>	Brain Research through Advancing Innovative Neurotechnologies (Project)
<i>Caltech</i>	California Institute of Technology
<i>Ce</i>	Caenorhabditis elegans, a simple organism (1000 cells incl. 302 neurons)
<i>CfC</i>	Closed-form Continuous-time Neural Networks
<i>CI/CD</i>	Continuous Integration / Continuous Delivery
<i>CLI</i>	Command Line Interface

-
- CNS* Central Nervous system
- CPU* Central Processing Unit, a central processor
- CTM* Computational Theory of Mind
- DevOps* Development and Operations, methodology representing a collaborative approach to performing company's application development and IT operations tasks
- DSS* Data Science Studio
- DT* Decision Trees
- ECG* Electrocardiogram
- ECR* Amazon Elastic Container Registry
- ECS* Amazon Elastic Container Service
- ELM* Extreme Learning Machine
- EPFL* École Polytechnique Fédérale de Lausanne
- ESM* Echo State Machines
- ESN* Echo State Networks
- EU* European Union
- F – MNIST* Fashion-MNIST dataset of Zalando's article images, each a 28x28 grayscale image with a label from 10 classes
- GB* Gigabyte, IEC 80000-13 confirms that a gigabyte is 10^9 bytes
- GBT* Gradient Boosted Trees Algorithm
- GENESIS* GEneral NEural SIMulation System
- GNU GPL* GNU General Public License
- GPIO* General Purpose I/O, an interface available on most modern microcontrollers
- GPU* Graphics Processing Unit, originally designed to accelerate computer graphics workloads
- GUI* Graphical User Interface

- HBP* Human Brain Project
- HDMI* High-Definition Multimedia Interface, a proprietary interface for transmitting compressed or uncompressed audio/video data
- HH* The Hodgkin-Huxley neuron model
- HPC* High Performance Computing
- HPI* Hasso Plattner Institute
- IA64* Intel Itanium Architecture-64 (Itanium microprocessors)
- IAM* AWS Identity and Access Management
- I&F* The integrate-and-fire neuron model
- I/O* Input/Output
- IoT* The Internet of Things, a network of (any) devices connecting and exchanging data with other devices and systems over the internet
- IP* Internet Protocol address
- IPC* Inter-process communication
- IT* Information Technology
- IZ* The Izhikevich neuron model
- LARS* Least Angle Regression
- LightGBM* Light Gradient-boosting Machine
- LSM* Liquid State Machine
- LSTM* Long Short Term Memory
- LXC* LinuX Containers
- MAUC* Multi-class Area Under the Curve
- MB* Megabyte, a common unit of digital information equal to one million bytes
- ML* Machine Learning

MLP Multi-layer Perceptron

MNIST Modified National Institute of Standards and Technology database

MPI Message Passing Interface

NALSM Neuron-Astrocyte Liquid State Machine

NCC Neocortical columns, also known as hypercolumns

NIC Network interface card, a hardware component allowing computer to connect over a network

N – MNIST Spiking version of the frame-based MNIST dataset

NSC Neural Simulations Cluster

NSP Neural Simulations Pipeline

OS Operating System

PDU Power Distribution Unit, a device that provides multiple outputs to distribute electric power to racks of computers or network devices in a data centre

PGENESIS Parallel extension to GENESIS

PID Process ID

POSIX Portable Operating System Interface

PVM Parallel Virtual Machine

R53 Amazon Route 53, a Domain Name System service

RBAC Role-based Access Control

R&D Research and Development

RDD Resilient Distributed Dataset

RNN Recurrent Neural Network

ROC Receiver Operating Characteristic curve

RPi Raspberry Pi, a complete single board computer manufactured in the UK

-
- RPr* ROCKPro64, a complete single board computer manufactured in Hong Kong
- RU* Rack Unit, a standardised form of servers: 2U is 1.75" x2 = 3.5 inches. All rackmount servers are 19" in width.
- S3* Amazon S3, a name for AWS Simple Cloud Storage service
- SBCC* Single Board Computer Cluster
- SD* Secure Digital, a non-volatile flash memory card format from the SD Association
- SGD* Stochastic Gradient Descent Algorithm
- SI* The International System of Units (from the French "Système international d'unités")
- SLFN* Single Hidden Layer Feedforward Networks
- SLI* Script Language Interpreter (of GENESIS)
- SNN* Spiking Neural Network
- SoC* System on chip
- SP* Separation Property of LSM
- STDP* Spike Timing Dependent Plasticity
- STPU* Spiking Temporal Processing Unit
- SVM* Support Vector Machines
- TDD* Test Driven Development
- TUI* Text-based User Interface
- URI* Uniform Resource Identifier, as defined in RFC 2396
- V1* Primary visual cortex
- VM* Virtual Machine
- VPC* Amazon Virtual Private Cloud
- WiFi* Wireless Fidelity, derived from the name of Wi-Fi Alliance, who proposed a number of wireless network protocols based on the IEEE 802.11 standards
- XGBoost* eXtreme Gradient Boosting Algorithm

Chapter 1

Modelling Networks of Spiking Neurons

1.1 Introduction

In the first chapter of this PhD dissertation the author describes why the brain simulations are critical for neuroscience. The chapter explains the *necessity and foundations of brain modelling and neural simulation*. We start with a description of how biological neuronal networks are represented as computational models. We continue with an introduction of our selected simulation platform GENESIS, which we will use as simulation engine in Neural Simulation Pipeline (NSP) to execute our Liquid State Machine (LSM) models on high-performance computing resources at Hasso Plattner Institute (HPI) in Germany (IA64), as well as self-built computational cluster made of Raspberry Pi 4B and ROCKPro64 nodes (ARM64). We follow with a characterisation of the limitations of the traditional workflow. We then report on the current state of the art in the simulation frameworks and tools. Finally, author concludes with a description of high performance computations and its general challenges.

The chapter is organised as follows. This Section 1.1 provides a discussion of this chapter. Section 1.2 explains the need for neural computations, whereas Section 1.3 focuses on how such artificial neural networks are (1) represented, (2) simulated using computational models on High Performance Computing (HPC) resources. Section 1.4 highlights why it is important to introduce a pipeline into the simulation of neural networks. Section 1.5 describes the limitations of the traditional approaches to neural simulations, and challenges to pre/post-experimental management of neural data. Section 1.6 explains how a visual cortex and hypercolumns work, whereas Section 1.7 focuses on key computational models of biological neuronal networks. Section 1.8 presents key challenges and progress in simulation of these models. Section 1.9 describes the existing simulation engines for large-scale neural networks. Section 1.10 introduces GENESIS, the key simulation engine that author uses

in this thesis. Finally, Section 1.11 focuses on the gaps in setting up and executing neural experiments (e.g. developing models, and setting up model parameter values), and the need to transform the end-to-end simulation development and execution workflow from one that is non-universal and manual to one that is standardised and at least partially automated.

As a result, we are getting a step closer to allow scientists to be able to perform a what-if analysis supported by machine learning algorithms, and/or to apply Big Data analytics to surface hidden trends, unknown correlations, or other meaningful insights, based on more reliable experimental data.

1.2 Problem, Motivation, and Proposed Solution

Computer-based simulations of neural networks help to overcome a key challenge in neuroscience, that is, to help us understand the joint behaviour of large groups of different types of neurons [14]. At the moment, we still have a limited knowledge about how the brain's neural networks work together to form complex behaviours such as visual signal understanding for action selection, motor learning or emotion. To overcome that and get a better understanding of information processing in the brain, we are currently able to simulate large neuronal network models that contain millions of neurons (10^5) and billions of synaptic connections (10^9) [16, 156, 4].

Although progress is constantly being made, even if we only look at the number of neurons (and their types) and their synaptic connections, we clearly see that we are orders of magnitude from the adequate representations of the biological systems, and still far from covering the complexity of the human brain. The present computer hardware simply fails to keep up with the increasing computational requirements of these simulations [138, 125].

New methods of computing are needed to meet the large computational requirements of brain scale networks. New advancements in simulation technologies must be achieved to allow for both the numerical models to leverage the maximum compute capability of various computing hardware in clusters efficiently. We should also start leveraging both the low power edge devices, GPUs, as well as recent neuromorphic architectures like Intel's Loihi [48] or SpiNNaker [74], a million-core computing engine built of an array of ARM9 processors designed to simulate the behaviour of up to a billion neurons in real time.

The new more complex brain models will likely require even higher performing I/O subsystems, as well as faster networks to avoid delays in moving large amounts of data for processing or analysis. The increasing compute-I/O gap has caused the traditional post-experiment analysis methods (related to batch processing) to fail to adequately scale to meet the new requirements [5, 90, 116].

Apart from that, new neuromorphic computational paradigms and hardware architectures emerge, and they can play an important role in running large scale brain simulations, delivering unprecedented computing resources, accompanied by limited power requirements [167].

This might open new ways for neural simulations to use new, distributed or on-edge low-power systems, and at the same time allow to create some new, more intelligent adaptive systems, analysing data and interacting with the environment in real-time.

From the neural simulation perspective though, it seems that we need new interactive pipelines and adaptive workflows to more actively analyse/monitor data points for their points of generation. Such an approach will reduce the need for data movement and it will allow to perform hypothesis-driven “what-if” analysis in near real time.

Neural simulation environments are often complex and expensive to build and maintain. Their configuration and model deployment might present a significant barrier for many researchers tackling biocybernetic modelling. This is because building, testing and deploying cybernetic models often requires different software libraries as dependencies, and the ability to consume significant amounts of parallel or distributed computing power to speed up the long running computations. The matter is especially important for developing larger, and more complex scripts simulating elements of human brain. The task is not trivial from technical perspective, even when using a well established simulation engine like on the General NEural Simulation System (GENESIS) [44].

In this thesis, I propose a Neural Simulation Pipeline (NSP) to facilitate the simulations of large, biologically plausible neural network models using GENESIS simulation engine. The NSP enables the integration of a more robust model development methodology with adequate testing. It also closes a gap in setting up, and executing the experiments across different execution environments incl. a high performance computing environment. In the later chapters, I assess the impact of running experiments using different neural networks on a HPC class system, as well as a self-made cluster built of Raspberry Pi and ROCKPro64 nodes, and in-cloud.

1.3 Simulating Neural Networks on HPC Resources

The human brain is complex. It is built of 100 billion interconnected neural cells forming an extremely large and complex network. In spite of all the recent developments in computing technology, it is still impossible to achieve the simulated neural networks at this scale. The current brain exploration methods are also limited. Scientists are prevented to insert electrodes into human brains due to ethical reasons. Even if this form of direct experimentation on the live human brain was possible, attaching an electrode to a single neural cell would be

technically problematic, as it would surely damage the cell (alongside with all its surrounding cells).

Therefore, computational models of neural networks are often the primary mechanism for scientific discovery in brain related sciences. This creates demand for larger scale and higher fidelity models. Such models, of different scale, can now be simulated using various computer hardware and software technologies, allowing for the modelling of small sub-cellular processes, up to the simulation of the complete functional areas of the brain on massively parallel computing environments [116].

We simulate models of neurobiological processes to validate the theory with observable results, and to investigate the relationship between their structure and function. Moreover, neural models are used to study dynamical systems for which any experimental data is not available.

The primary challenge for neural simulation is that the current brain models have to be significantly scaled down. The number of neurons and synapses in these models is limited by the number of network elements that can reside in the computer's memory at any one time [125].

Therefore, to overcome this limitation neuroinformatics focused on data structures to enable the simulation of neural network models at increasing sizes [39, 51, 138, 116, 98]. Apart from the challenge of representing a large numbers of network elements in computer's memory, the time needed to instantiate a network is also concerning [116]. As models become larger, the computational requirements and time to instantiate them grow, which has the potential to affect the simulation performance negatively.

Parallel neuronal simulators have been developed that partition the neural network across multiple processing elements to speed up the computation. By default the processing of spiking neural networks (SNN) should be accelerated with multi core central processing units (CPU), rather than graphics processing units (GPU). This is because of CPU's larger flexibility of task and thread level parallelism. SNN simulations could be less efficient on GPUs, as the high GPU efficiency for traditional artificial neural networks (ANN) relies on fixed arrays [18]. Moreover, as GPUs tend to offer lower precision, modern applications with higher accuracy needs often improve performance and satisfy these accuracy requirements by implementing so called "mixed precision computations" [135], that are not implemented for the well established neural simulation engines [215].

To summarise, given the scale and complexity of brain-like neural networks, their simulation often requires powerful supercomputers (and complex simulation setup). These large computers, often need to also consume a lot of electricity, are expensive to build and maintain,

and are not easily available for these who would need to use them (there are clear entry barriers to the neural simulation).

In this thesis, I analyse the impact of execution environment like traditional HPC resources (e.g., single fat nodes and high-end clusters), and low-end, low power single board computer nodes on performance, data generation, and science delivered by GENESIS for increasingly large and complex models of the brain's visual cortex. I explore the relationship between model fidelity (i.e., the level to which our model successfully reproduces the behaviour of neuronal networks in nature) and performance, by increasing the number of cells simulated in two models, and I run the simulations on both high-end (HPC), and a self-built, low-end cluster, as well as in public cloud.

1.4 Integrating Pipeline into Neural Simulations

Constant evolution of the simulation technology have enabled improvement in the fidelity and sizes of the neural network models [2]. As a result these models generate more data [139].

A common scientific workflow in neuronal simulation models is to analyse the simulation results *after* completing the simulation. As already mentioned in Section 1.3, these workflows are often happening in a high performance computing environment, where simulation time is expensive and scarce. A typical workflow includes preparing simulation input, configuration of model parameters, and execution of the computations. This is concluded by the post-hoc analysis and/or visualisation of the simulation data. These post-processing steps are usually executed in sequences [246, 53, 196].

As a result, the large amount of data produced by more complex, higher fidelity models are becoming more and more difficult to save and transfer for the post-hoc analysis. A possible solution to this problem is to adjust the scientific workflow to allow for more automated and interactive experiment management. In this thesis, I design a neural simulation pipeline. I evaluate the scalability of the system, with the intention of presenting not only performance, but also the execution cost in-cloud.

Da Cruz et al. [46] show that despite the high interest in scientific workflow management systems, it is still “an open field” due to its complexity. Authors suggest promoting the interoperability of such systems, so that scientists are able to manage the data from executions of distributed heterogeneous workflows more efficiently.

I attempt to investigate the relationship between experiment setup and simulation run in computational neuroscience. This is to improve experiment management, and facilitate the analysis of data generated by simulation of the electrical activity of a brain's neuronal network.

1.5 Limitations of the Current State of the Art (EEG)

At present, brain electrical activity is measured using electroencephalography (EEG). This method uses electrodes that are placed on the outside of the head to record the activity of all the neurons simultaneously. EEG is a well established method in neuroscience. It is also the most commonly used method to study brain pathology [130].

It is believed that the brain's cognitive functions are resulting from the neuronal connectivity, and the electrical activity of neurons' multitude of connections. This activity allows for information processing that is achieved by the synchronisation of the spike trains of large networks of neural cells [97, 25].

When the ensembles of neurons are active, EEG measures and records the differences of the ionic current in the brain [69, 145].

EEG measurements cover vast amounts of information about the activity of the brain. There are many approaches for analysing this information, most of them based on different spectral analysis methods, usually measuring so called spontaneous electrical activity of the brain over a specified time [69]. The activity often has oscillatory characteristics, with frequencies ranging from 1 to 100 Hz, and it is associated with human cognition [144].

When working with EEG signals, the main challenge is that such signals are non-deterministic. As such they are not restricted to special formations, in the same way that electrocardiogram signals (ECG) are. The result of this lack of restrictions is that the analysis is performed using the parametric and statistical approaches (such as e.g. time-frequency analysis) [123]. Unfortunately none of these approaches allows us to observe the detailed neuronal connectivity, and how that impacts the activity of the the whole brain [4].

In order to address this challenge, larger and more realistic brain models of brain biophysical activity have been created. Such models, which are more biologically accurate, are significantly more computationally intensive, both in numbers of required neurons (e.g. the cerebral cortex contains 14 to 20 billion neurons [100]), as well as their synaptic connectivity (e.g. the cerebral cortex is connected with trillions of synapses).

1.6 Visual Cortex

1.6.1 Visual Cortex as Part of Cerebral Cortex

The *cerebral cortex*, forming approximately 80% of the human brain, is made of so-called *grey matter*. This part of the brain is responsible for all the key brain functions, such as perception, communication, adaptation to new environment conditions, as well as complex

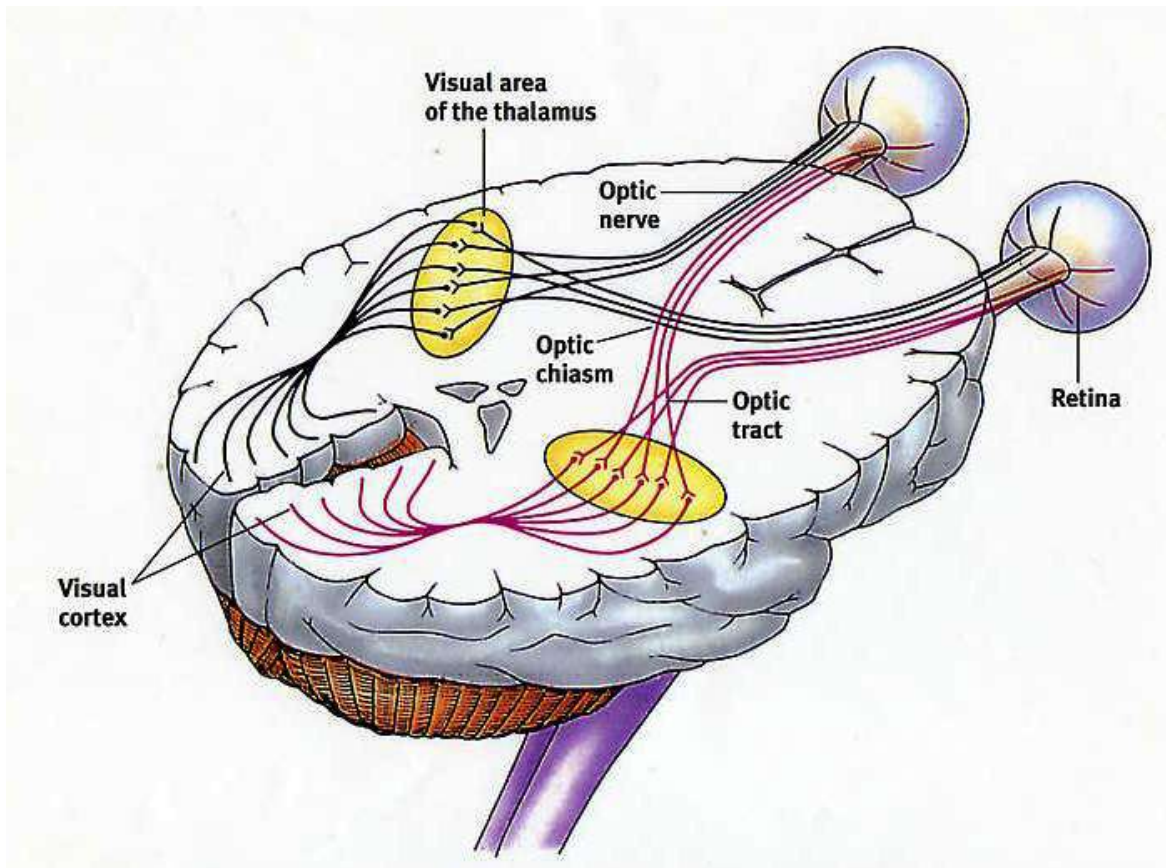
cognitive functions of planning, and higher level mental functions [213]. The most evolutionarily advanced part of the cerebral cortex is called *neocortex*. It covers 90% of the volume of the human cerebral cortex, and it appears not earlier than in mammals [100].

Cognitive abilities depend largely on the brain's ability to anticipate future events in order to react appropriately in a changing environment. The cerebral cortex with the neocortex uses complex brain networks that transform sensory stimuli into a multidimensional response in real time [200]. The signals received from the body sensors are mapped in both the neocortex and in the sub-cortex, as well as into the sub-cortical region. Within the cortex itself, information is repeatedly duplicated to maintain a coherent and complete picture of the situation. This is fuelled by the memory of past experiences. There are well known topographic maps of the neocortex that accurately specify the general basic function of each brain region and show how all the cortical regions are internally connected [213]. The key visual processes in the cortex that I focus on happen in the occipital lobe, so 'in the back side' of the head, as per Fig. 1.1.

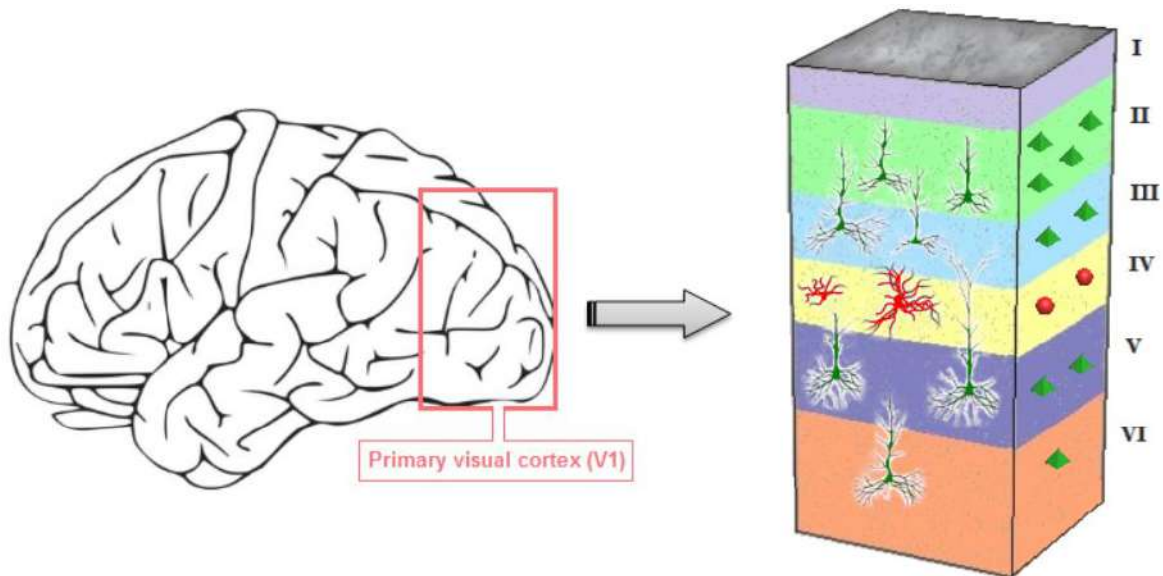
The whole visual process starts with retina passing the electrical signals via the optic nerves, and later so called visual pathways to the primary visual cortex, as shown in the Fig. 1.1. Visual area constitutes about 25% of the cortex in humans with approximately 5 billion neurons. The visual cortex can be split up further into 5 layers. These are V1, V2, V3, V4 and V5. V1 is often referred to as primary visual cortex. The studies have revealed that these visual regions differ on the basis of their anatomical architecture, topography and physiological properties [175, 11]. Each region is involved in processing a multitude of information related to shape, orientation, colour, movement and size of objects. The information results from the visual pathways, transforming an image applied to a retina.

The images we see with our eyes are processed in the primary visual cortex. When we see an object, the image that is captured by our eyes is always inverted (back to front) and blurry. Our brains are equipped with enormous processing power, allowing real-time transformations that correct the received image. This processing involves the recurrent cortical circuits [157].

As stated by Tong [216], V1 is probably the most well known area of primate cortex, but whether this region contributes directly to conscious visual experience is still controversial. Initial studies found out that visual awareness was best correlated with neural activity in extrastriate cortex visual areas, frontal lobe areas, such as the frontal eye field. More recent studies have found similarly powerful effects in the V1, with interactive models proposing that recurrent connections between V1 and higher areas form functional circuits that support awareness [170].



(a) A labelled side view of the visual pathway in human brain [61].



(b) The grey matter in the primary visual cortex is divided into six layers that comprise of pyramidal cells and interneurons [11].

Fig. 1.1 Visual system in humans, a view of visual cortex and the pathways.

Similar to other areas of the brain, the neurons in the visual cortex are connected to one another by synapses [175]. Neurons contained in the same micro-column share the same static and physiological dynamic properties [170].

The excitatory neurons in the visual cortex are of a great significance for us in understanding brain functions. Such neurons use the excitatory neurotransmitters to fire a signal called an action potential into the receiving neuron. In contrast, the inhibitory neurons cause neighbouring connected neurons to be less likely to fire. 2/3 of neurons are organised into sub-networks of pyramidal neurons [175].

Finally, it is widely understood that it is the *connectivity* within the brain that is responsible for the cognition, perception, decision-making, and language [27, 188, 202].

1.6.2 Hypercolumns as Information Processing Units

A fundamental characteristic of the cortex is its repetitive structure. Billions of neurons are connected to their neighbours, forming a unique 'wiring' of the neuronal network of our brain. These connections determine both the individual uniqueness of each person, as well as form some general connectivity patterns, as recorded by Brain Activity Map Project, aimed at reconstructing the full record of neural activity across complete neural circuits [4].

These neuronal connections are neither random, nor arbitrary. Neurons form multiple, small computational columns of a cylindrical shape throughout the cortex. They are made of a complex network of around 10000 connected neurons. Their size differs. As a way of example, the dimensions of an ocular dominance column in the cat is 0.5 mm, whereas cat's orientation column size is 0.1 mm. Although in both cat and monkey large variations in the sizes were observed, a base diameter of a column can be approximated to 0.5 mm and a height of up-to 2 mm [6].

There are millions of microcircuits in any mammalian cortex. Their exact number differ between different species, but the difference between a rat and a human brain is mainly in the number of microcircuits that each brain is built of. The biological computational columns resemble microcircuits in a computer processor, which is why they are often called *neural microcircuits*, or (due to their cylindrical shape) *neocortical columns* (NCC) or simply *hypercolumns* [210].

There is a consensus that information processing specific to the neocortex is performed by neuronal microcircuits. Despite their identical structure that is repeated several times, their functions can be radically different depending on the position in which these basic information processing units are located in the brain. Moreover, multiple microcircuits may form a group of units, specialising in a certain task. Moreover, the same microcircuit may

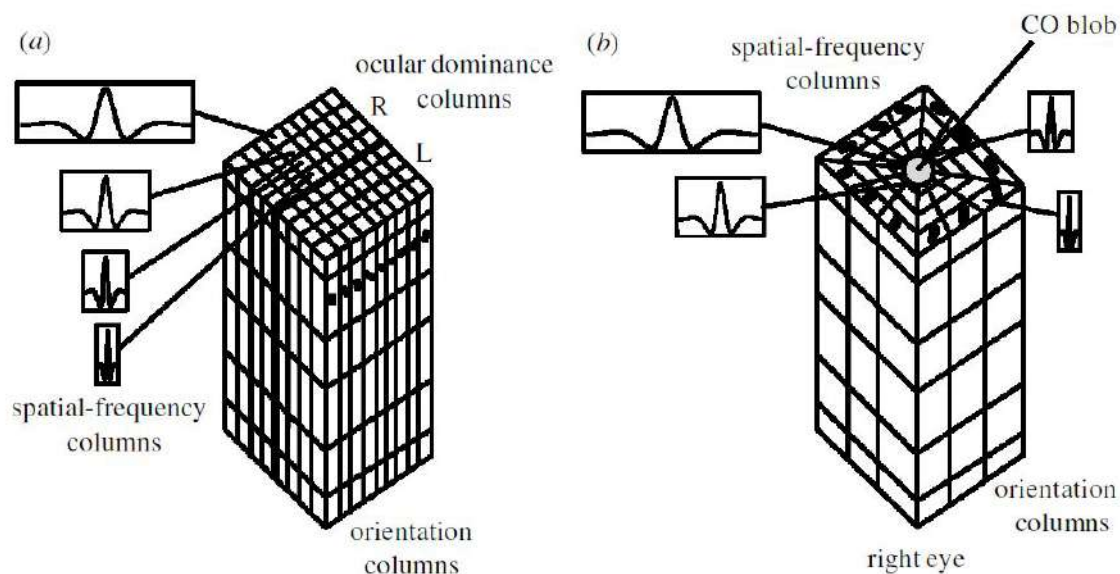


Fig. 1.2 (a) The De Valois and De Valois' icecube model of a cat V1 hypercolumn. (b) The modified icecube model for macaque's V1, as presented by Bressloff [28].

belong to a few different groups, and respond (or ignore) signals in the cortex depending on the dynamic situation in the brain.

The microcircuits themselves, and all other neurons in the layers of neocortex represent the incoming, changing states of the environment in the form of impulses. The anatomical and physiological properties, as well as the probability of connection between neurons are unique for each synaptic connection. The advantages of such a recurrent system are enormous. It exhibits much greater information processing capabilities than a simple random recurrent network [152].

Based on the studies of different mammals, including cats and macaques, it has been shown that certain columns recognise the slope of a line (these are called orientation columns), while other like these represented in the diagram by a cylinder (blob), play an important part in colour recognition. That means that the mammalian V1 cortex has the spatially periodic micro structure of the cortex [28]. The alignment of microcircuits within a fragment of mammalian visual cortex is shown in Fig. 1.2.

This leads us to conclusion that it is the distribution of the microcircuits that allows the cortex to cope with different tasks of high complexity, and the same information processing unit, but located in different brain region, can perform completely different operations.

1.7 Simulating Spiking Neural Networks

From a computational perspective, neural networks can be defined as directed, distance weighted graphs. In these graphs neurons form nodes, whereas edges represent synapses [183]. In such a network neurons communicate by passing spikes of action potentials [183]. This communication happens (1) in a single direction, from the node sending the spike to the node receiving the action potential, (2) with a specified delay. Each neuron also has a defined threshold. If this threshold is exceeded as a result of receiving one or more inputs from connected cells, the neuron will fire, and therefore will continue the propagation of the signal across the network [107]. We present the three most important computational models of neurons used in simulations.

1.7.1 Hodgkin-Huxley Model

The most important and popular computational model of neurons was presented in 1952 by Hodgkin-Huxley [107], who were later awarded a Nobel prize for their contribution. Their mathematical model of neuron was based on the measurements of the giant axon of a squid, that allow them to describe the electric current flow through the surface of axon's membrane. As a result, they were able to build an artificial neuron that was able to realistically simulate the initiation and propagation of the action potential in a neuron, and recreate the behaviour of a biological system through a human built electrical system.

In their model the neuron's membrane acts as a capacitor. It separates the interior of the cell from the outside liquids, allowing the cell to store electrical charge.

They defined electric conductance g_{ion} as being proportional to the number of open channels. The net movement of ions across the cell membrane is defined as the ion current. If we denote V as the membrane potential and V_{ion} the reversal potential, than using the Ohm's Law, the electrical current can be expressed as:

$$I_{ion} = g_{ion}(V - V_{ion}) \quad (1.1)$$

The Hodgkin-Huxley model considers three ion channels that are the voltage-dependent K⁺ and Na⁺ channels, as well as the static leaky channel. The channel can be expressed by equations:

$$g_K = \bar{g}_K n^4, g_{Na} = \bar{g}_{Na} m^3 h \quad (1.2)$$

where \bar{g}_K and g_{Na} the maximum conductance of respectively the sodium and potassium channels, with the leakage channel considered to be constant, and modelled by a given resistance. The variables m and h describe the activation and deactivation of the Na⁺ channel respectively, while n describes the activation of the K⁺ channel. They are called *gating variables* and control the openings of ion channels, that can be interpreted as a molecular switch between two states with voltage-dependent transition [159].

The dynamics of the model is expressed with the three fundamental differential equations:

$$\frac{dm}{dt} = \alpha_m(v)(1 - m) - \beta_m(v)m \quad (1.3)$$

$$\frac{dn}{dt} = \alpha_n(v)(1 - n) - \beta_n(v)n \quad (1.4)$$

$$\frac{dh}{dt} = \alpha_h(v)(1 - h) - \beta_h(v)h \quad (1.5)$$

Hodgkin and Huxley formulated the above equations and their parameters to fit the experimental data that they had gathered during the measurements of the giant axon of a squid. The α and β determine the evolution of the model variables. The model is described in more detail in [201]. As it can be represented as an electrical circuit it is often referred to as a *conductance-based* model, to differentiate from other models of neurons. I present the model in more detail in Chapter 1.9 of this thesis, where we explain how the simulation of a standard GENESIS neural compartment works.

To summarise, the Hodgkin-Huxley model, in spite of its age, remains the fundamental model for any biologically representative neural simulations. It exhibits all the computational properties of spiking and bursting models. Unfortunately, the model is also the most computationally expensive to implement, and difficult to analyse. Therefore, it is generally used to simulate smaller groups of neurons, often with no constraints on simulation time (such models tend to run for a long time), often with expensive HPC simulation setups.

1.7.2 Leaky Integrate-and-Fire Model

Assuming the computational complexity of the original Hodgkin-Huxley model, new models were being created. The integrate-and-fire (I&F) neuron model is one of the most popular spiking models. It is often used in the study of the neural information processing, or to

analyse the neural systems. It is simple for mathematical analysis, and allows building large networks to analyse the network dynamics in neuron coding or memory [81]

In contrast to the HH model the IF model is represented by a simple circuit built of a resistor R and capacitor C driven by a current $I(t)$. The changes of the membrane potential V can be calculated using a *leaky integrator* differential equation of:

$$\frac{dV}{dt} = -\frac{V(t)}{T_m} + \frac{I(t)}{C} \quad (1.6)$$

This equation assumes that the sodium channel is not voltage dependent, so it models the neuronal effects of the sodium and leaky channels only. $I(t)$ represents the total incoming current from neurons prior to the receiving synapse.

If no new spikes arrive to the synapses, the potential V reduces to the resting potential. This happens with a speed indicated by membrane time constant $T_m = RC$. As soon as the threshold voltage is reached, the stimulated neuron fires an output spike.

As a result of this firing V is changing to V_r (the rest potential) for the so called *refractory period* of T_{refrac} , where t_0 is the time of firing. At the end of the process $V(t)$ changes again (due to the newly received synaptic current) according to Equation 1.6. The whole process can be described by the below Equation 1.7:

$$V(t) = V_r \text{ for } t_0 < t \leq t_0 + T_{refrac}. \quad (1.7)$$

1.7.3 Izhikevich Model

The other important model of neuronal networks was presented by Eugene Izhikevich [118]. The Izhikevich (IZ) model is computationally simple, and able to simulate several neuronal responses. The model is based on two ordinary differential equations:

$$\frac{dV}{dt} = 0.04 V^2 + 5 V + 140 - u + I(t) \quad (1.8)$$

$$\frac{du}{dt} = a (b V - u) \quad (1.9)$$

The IZ model uses post-spike resetting, defined as:

$$\text{if } V \geq +30 \text{ mV then } \begin{cases} V \leftarrow c \\ u \leftarrow u + d \end{cases} \quad (1.10)$$

The variable u denotes *membrane recovery*. It models the activation of K^+ and inactivation of Na^+ ionic current flows. It also provides a negative feedback to *membrane potential* V . Both V and u are reset after the neuron spike exceeds a threshold of 30 mV, according to the Equation 1.10.

The other parameters a , b , c , and d can be assigned different values depending on the type of neuron being simulated: a refers to the time scale of u ; b to the sensitivity of u reflecting the variations of V ; and finally c refers to the post-spike reset value of V related to high-threshold K^+ conductance; whereas d refers to the post-spike reset of u due to slow high-threshold K^+ and Na^+ conductances.

As stated by Izhikevich [119] in his later work, the firing patterns of all known types of cortical neurons can be reproduced with the choice of parameter settings in the model [118] summarised in this sub-chapter.

1.8 Challenges and Progress

In spite of recent progress in neuroscience, the understanding of how neuronal connectivity allows us to perform advanced actions such as understanding of what we see, hear and feel; or complex cognitive activities, such as planning actions in advance, is still poorly understood. Improved understanding of our brains allows us not only to understand how we think and what the biochemical processes in our brains are, but it also contributes to the creation of new, more efficient computing paradigms, such as neuromorphic computing [41].

The main challenge in science focusing on the brain is that a direct observation of neuronal activity of individual neurons, or their groups, is currently not technically possible using the state-of-the art *in-vivo* human neuro-imaging. As these methods are deemed often inefficient, a focus shifted to brain simulations [4].

To achieve large scale simulations of a brain-like neural system having approximately 10^9 neural cells and 10^{12} synapses, much larger neural networks models are needed, and these models are not currently available. At the moment the computational resources needed for simulation of these large neural systems exceed the available capacity of small-to-medium sized HPC setups, which are traditionally made available to scientists in their own research institutions. As a result, these growing computational demands of brain researchers have been

targeting purpose-built neuromorphic hardware that relies on massively parallel computing architectures like SpiNNaker [167].

In order to start studying the fundamental information processing in the human brain, the general requirements for the simulated neural network size in computational neuroscience are models built of more than 10^5 neurons and 10^9 synaptic connections [116]. Such models allow scientists to explore the impact of different network models, as well as different synaptic connectivity. The last decade has brought an increased performance and capability of neural simulations, often using supercomputer hardware, or large HPC clusters [39, 51, 99, 98, 187]. The developments have been revolving around simulating models that are around the size of a cubic millimetre of brain cortex (so approximately 10^5 neurons), using small to medium sized HPC clusters. A major challenge in these simulations were large numbers of synapses connecting neurons [125]. This was the challenge because the simulation of the realistic number of synaptic connections has often exceeded the available memory capacity of the HPC hardware that was made available to the brain scientists. In cases when the large memory resources were available, the subsequent challenges were the long simulation time and inability to analyse/optimize model parameters [138].

1.9 Current State of the Art in Simulation Frameworks

Computer simulation of neuronal networks has established itself as a major method of the scientific analysis in neuroscience. The simulation has been used to study how the anatomical observations relate to the physiological information gathered. Simulation is also used to research dynamic systems that are difficult to investigate with the current analytical methods. The first software simulators and neural models designed to study complex neural systems became popular among scientists at the end of 1990's. Most of the analyses of the brain's neural networks were done on data from the partial models of the brain of a monkey [68], cat [189] and rat [37]. At this point neuroscience realised that collecting and studying experimental data was not enough to explain how neural circuits and columns work. This realisation stimulated a more quantitative approach to study of neuronal networks, and refocused scientists to build computational models to study network dynamics. As a result, a new scientific domain was created that is called computational neuroscience [63].

The first significant computers-based network model of the whole neural system was simulated with a model of the *Caenorhabditis elegans* (Ce) [200]. The Ce is a primitive, free-living transparent nematode, that is about 1 mm long and lives in soil environments. The organism has been studied intensively by the research group of Sydney Brenner since 1963. It was selected due to its simplicity, as the whole organism consisted of only about 1000 cells

and 302 neurons. This level of complexity was ideal for researching information processing in neural networks [225].

The next jump in the simulation size came from the successful modelling of a full cortical column of a rat brain [37]. As mentioned in Section 1.8 the recent notable advancement in the simulation neural networks has come from the BBP and their simulation of the rat's cortical mesocircuit (100 neocortical columns), that was followed by the model of the whole rat brain [156] in 2014. This progress cannot be undermined, although the whole HBP was not developing without challenges and scientific critic [237].

As mentioned by Gaute T. Einevoll et al. [65] the HBP project has overall contributed to the development of high-quality brain simulators like NEURON [39] and NEST [82]. Authors suggest that the infrastructure for brain simulations require long-term funding, similar to that which was delivered to other branches of science in which many scales have been bridged, listing the numerical weather prediction and the engineering underlying cellular telephones as examples. It is not only the computational neuroscience that has been almost totally dependent on mathematics and simulations to bridge models at different scales; therefore as they [65] state “a natural question is, do we have a chance of ever understanding brain function without the brain simulations?”

1.10 Simulation System

1.10.1 General Neural Simulation System

GENESIS [23, 86] is an object-oriented multi-function neural simulation software package that allows scientists to flexibly build high-fidelity neurobiological models. These models are capable of simulating brain functions on different levels, from the level of small sub-cellular components to sophisticated large and complex neural networks.

GENESIS was originally developed in 1989 by Dr. James M. Bower in his laboratory at California Institute of Technology (Caltech). GENESIS is designed to be easily extensible and adaptable to run on different HPC clusters [22]. Apart from its maturity, one of the key advantages of using GENESIS is its openness. The software is distributed under GNU General Public License (GNU GPL), so its users have the freedom to run, share and modify the simulation engine's source code, and any derivative work must also be distributed under the same or equivalent license terms.

There are two major flavours of the simulation engine. A standard GENESIS, and a PGENESIS that runs on a wide range of hardware using the Message Passing Interface (MPI) and the Parallel Virtual Machine (PVM) [21].

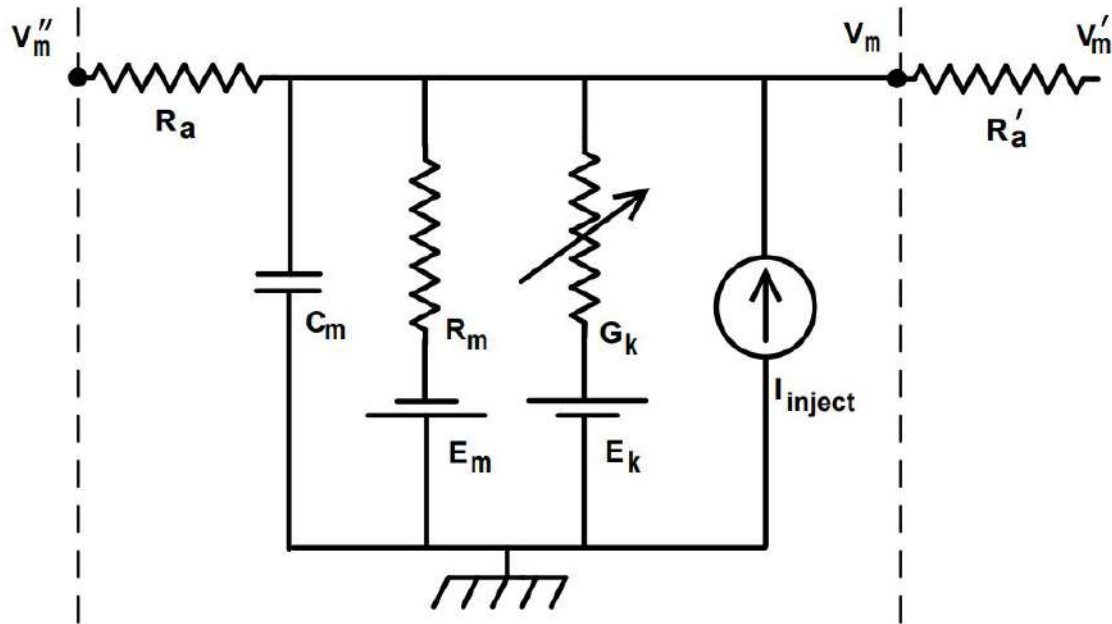


Fig. 1.3 The equivalent circuit for a standard GENESIS neural compartment.

GENESIS simulations are programmed using objects that are fed inputs, perform some type of mathematical operation on them, and then based on the result of the operation, generate outputs which are input to the other objects. Neurons in GENESIS models are built from these basic components in a *compartmental fashion* [15] using a GENESIS Script Language Interpreter (SLI) that provides the programmer with a built-in language to define and manipulate these GENESIS objects. In the compartmental approach, neuron's compartments in GENESIS are linked to their ion channels and the channels are linked together to form multi-compartmental neurons of up to 50–74 compartments per neuron. GENESIS simulations scale on super-computing resources to neural network sizes as large as 9×10^6 neurons with 18×10^9 synapses and 2.2×10^6 neurons with 45×10^9 synapses [44], which can form any required complexity needed.

Individual neurons modelled using this approach are able to generate realistic electrical activity in the 1-100 Hz frequency range. Their activity is computationally modelled through a method called Equivalent Circuit of a Single Compartment. Fig 1.3 shows the equivalent electrical circuit of a basic GENESIS neural compartment, as presented in the Book of GENESIS [23] on page 11.

Compartmental modelling of single neurons has been in focus for computational neuroscience since 1990s. The idea of an equivalent electrical circuit for a cellular membrane is the foundation of all compartmental modelling, and as such in modern neuroinformatics. This is

because neuronal membranes have been demonstrating the behaviour of simple electrical circuits with a given capacitance, resistance, and voltage sources. If used as model parameters they define passive properties that are responsible for the way that electrical impulses are transmitted along the neuron's dendritic tree. Since its beginnings GENESIS supported advanced cell modelling with a consideration for all the passive properties of a cell, as well as active properties provided by ligand-dependent conductances, or different voltage. In this way the software could be used to problematically model almost any structure, including even more recent composite models [43], using ligand-gated ion channels, voltage-gated, and voltage- and concentration-dependent conductances.

Fig. 1.3 shows the equivalent electrical circuit of a standard GENESIS neural compartment. V_m represents the membrane potential, or the potential inside of a neuronal compartment in relation to a point outside of the cell. The ground loop symbol on the base of the figure represents an external point with an exact zero potential.

The cell compartment acts as a capacitor, that is either charged or discharged by current flowing into or out. The flow may come several directions:

1. from the adjacent compartment(s)
2. from the passage of ions through channels in the cell membrane
3. from the current injected using an electrode inserted into the cell

The membrane capacitance is indicated by C_m . It can cause a current flow into or out of the compartment through the axial resistances R_a or R'_a when there is a difference in potential of $V'_m - V''_m$ between the two compartments.

The resistor (with the arrow) represents one of many possible variable channel conductances that are specific to a particular ion; their combinations, once set, can give each neuron its individual type, or as described in GENESIS documentation [15] its 'unique computational properties'. G_k indicates conductance, and as conductance is the reciprocal of resistance, it's measured reciprocal ohms, or siemens¹.

In a real neuron the difference in the concentration of the ion between the internal and the external points of the cell result in an osmotic pressure which moves ions along the concentration gradient. This results in a charge displacement that creates a potential difference opposing this flow. The membrane potential at which there is no net move of the ion along the concentration gradient is called the equilibrium potential E_k (or the reversal potential). It is represented by a battery sign in the circuit.

¹1 Reciprocal Siemens (1/S) is equal to 1 ohm, while 1 Ohm (Ω) = 1 ohm.

If no electrical input is provided to the cell V_m will be equal to a rest potential E_r (ranging from -40 to -100 mV).

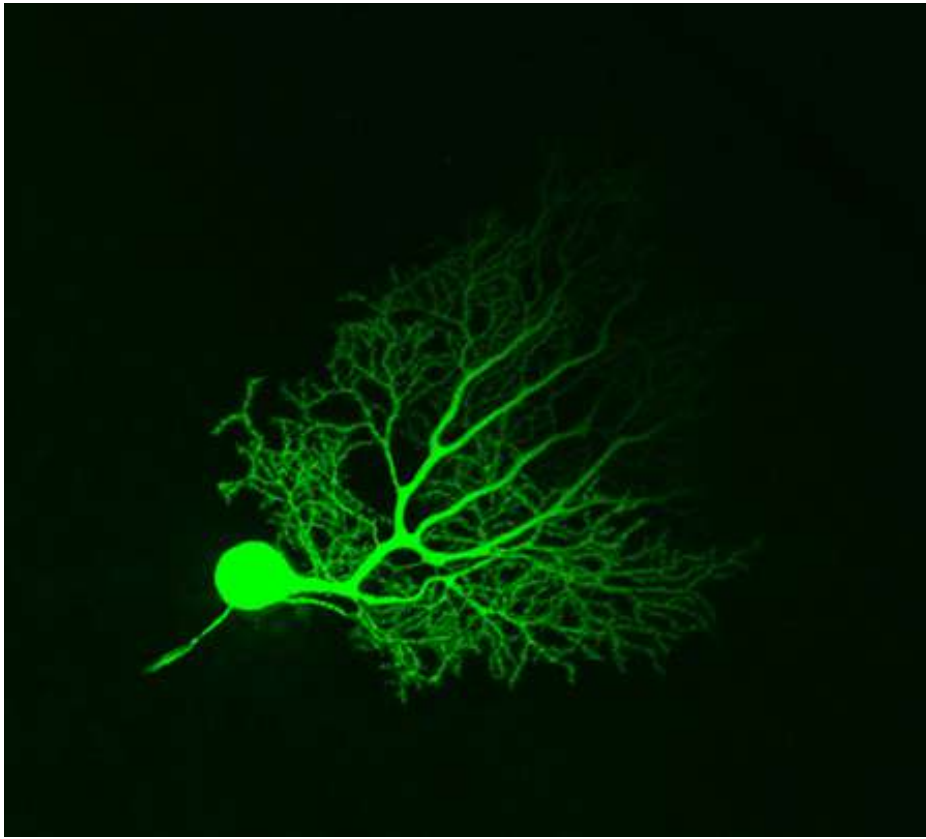
As described in the Book of GENESIS: Exploring Realistic Neural Models with the General NEural SIMulation System [23], the equivalent circuit has the other resistor and battery that link the exterior and the interior of the cell, and represent the combined effect of passive channels (mainly those for chloride ions). In biological cells they are having a relatively fixed conductance. This resistance is called the membrane resistance R_m , or a leakage conductance $G_l = 1/R_m$. E_m , the associated equilibrium potential is typically close to the rest potential. In simulations, it is given a slightly different value (E_l) in order to reduce the net channel current to zero when $V_m = E_r$. The current source I_i represents an optional injection current which could be provided by an electrode inserted directly into the cell compartment. The ability to perform this operation is a great advantage of neuronal simulations over the traditional, experimental approach. In a normal experimental setting, a neuron would surely get destroyed after any mechanical attempt of injecting the current directly into the cell compartment.

By defining the equivalent circuit for a neuronal compartment we can calculate a V_m at any point of simulation using a differential equation. This differential equation expresses the fact that the rate of change of the potential across C_m is proportional to the net current flowing into the compartment to charge the capacitance. What GENESIS than calculates is the current due to each of the sources shown in Fig. 1.3 using Ohm's law:

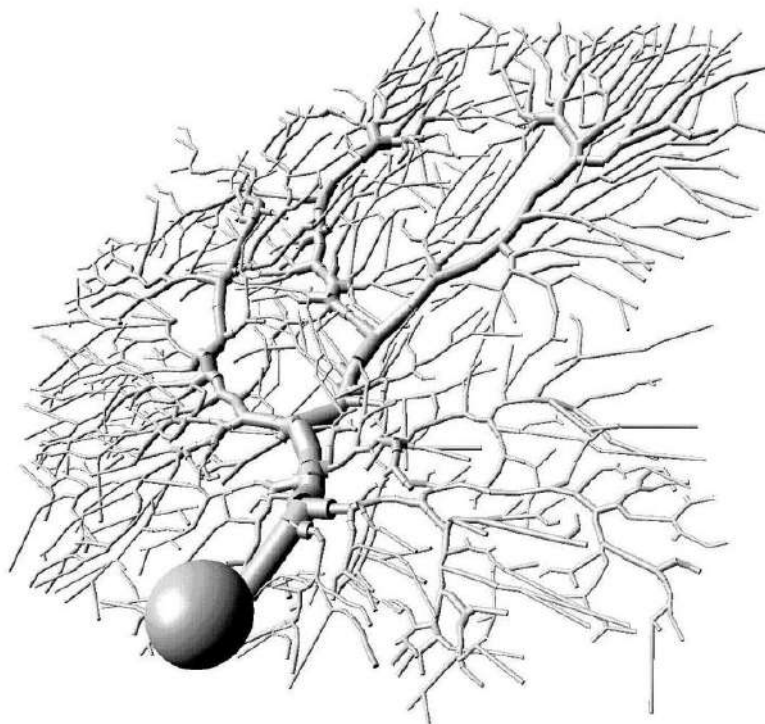
$$C_m \frac{dV_m}{dt} = \frac{(E_m - V_m)}{R_m} + \sum_k [(E_k - V_m)G_k] + \frac{(V'_m - V_m)}{R'_a} + \frac{(V''_m - V_m)}{R_a} + I_i \quad (1.11)$$

The models based on the compartmental approach can achieve very high levels of complexity, especially if we combine multiple cells together, to form more advanced structures. Fig. 1.4 presents a neuron and its high fidelity, multi-compartmental model. The same technique has also been successfully used in large simulations of the whole mammalian visual cortex [230] or to build a Liquid State Machine (LSM) [229]. The method forms a foundation, on which I build throughout this PhD thesis.

GENESIS employs a high-level simulation programming language to parameterise and build individual neurons and their networks through its Script Language Interpreter (SLI). The software can be used both interactively and in a batch mode [15]. GENESIS is written in C, the Portable Operating System Interface (POSIX) for its I/O processing, and can be configured to use either an X Window front-end called XODUS as its Graphical User Interface (GUI), or UNIX console for its text-based user interface (TUI).



(a) Purkinje neuron from mouse cerebellum injected with Lucifer Yellow and imaged using confocal microscopy [158].



(b) A detailed multi-compartmental model of a cerebellar Purkinje cell, visualisation by Jason Leigh using the GENESIS Visualizer program [23].

Fig. 1.4 A Purkinje cerebellar cell with dendrites, soma, and axon and its simplified discrete compartmental model built with GENESIS.

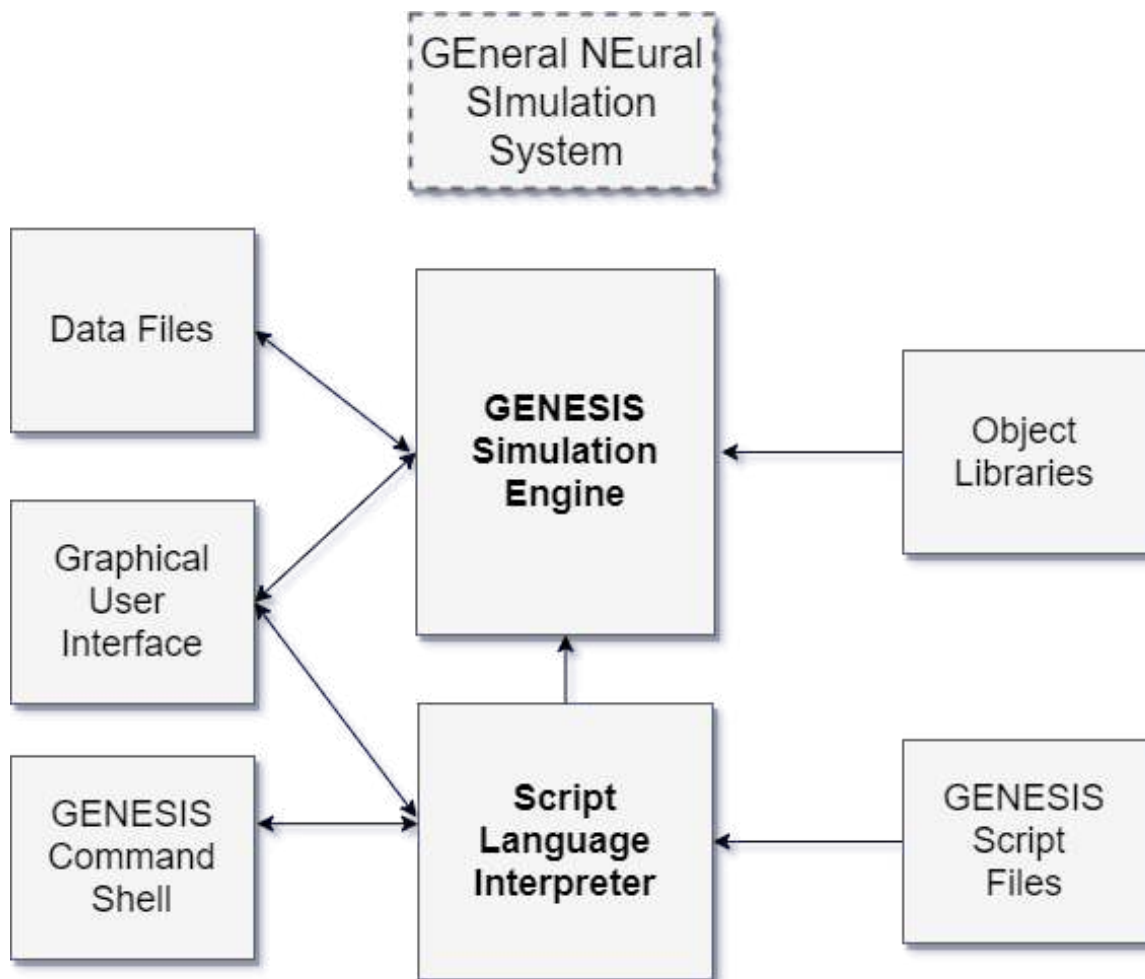


Fig. 1.5 GENESIS' high-level architecture.

Fig. 1.5 shows an overview of the high-level architecture and modular structure of the GENESIS neural simulator (i.e., scripting language interpreter and user interfaces used to configure and construct the neuronal networks). This overview has been recreated from the GENESIS documentation [23]. The workflow of a GENESIS simulation is broken into two main phases:

Model initialisation During the initialisation phase, GENESIS launches the model's input scripts. The scripts hold the initial status and values of the model. After parameters values have been initialised, the model proceeds to establish connections between neurons according to probabilities outlined in the initialisation scripts. Once all connections have been completed, the entire neural network is written to disk, if the I/O setting in GENESIS is set to on. This can be very time consuming.

Model computation Following writing the network out to disk, GENESIS enters the simulation phase. GENESIS runs for a predefined maximum amount of steps; solving underlying model differential equations and updating simulation variables at every time step. If I/O is on, GENESIS writes out performance and behavioural data to disk at each time step.

Moreover, GENESIS from version 2.3 contains Kinetikit, an interface and utilities for developing simulations of chemical kinetics. This extension contains a comprehensive graphical simulation environment for modelling biochemical signalling pathways using deterministic and stochastic methods [220]. The extended GENESIS becomes a tool to investigate also the *bio-mechanics of the brain* incl. its time-dependent temperature and pressure variations, or the liquid behaviours in contrast to ideal conditions. As such the GENESIS/Kinetikit simulations could be used to study the dynamics of cerebrospinal fluid flow and pressure, which can provide valuable information for diagnosing and managing fluid disorders or testing the effects of different interventions or optimising treatment strategies [171].

1.10.2 Other Simulation Frameworks

GENESIS is one of a few research tools that is able to simulate structurally realistic computations, delivered with large-scale parallel execution. It is recognised for its compartmental design that offers high fidelity of simulations. Nevertheless, it is one of a few recognised simulation tools that can be used to simulate SNNs [29].

Parallel simulations of spiking neural networks are also supported by BRIAN [87], NEURON [163], NEST [82], NCS [29], SPLIT [29] Nengo [17], and GeNN [236]. In spite of the existence of several tools, the research and development in parallel neuronal

simulation have been relatively modest in comparison to multiple frameworks in the area of e.g. (traditional) deep artificial neural networks. There has also been limited research with regards to the study of the impact of neural network parameterisation on performance and data generation, as well as computational steering [160].

Most publications in this area have focused on profiling the CPU communication, but have not considered Input/Output (I/O). Authors like Hines [102] look at how inter-processor spike communication affects total simulation time, by focusing on the spike exchange methods such as *MPI_Allgather*, or non-blocking *Mult-Send*. Shehzad [193] focuses on inter-processor spike communication to improve remote memory access for NEURON. This work analyses the impact of different computing resources on compute performance, data generation, and science delivered by the General Neural Simulation System (GENESIS) for increasingly complex models of Liquid State Machines.

Fig. 1.6 presents the relationship between experiment setup and simulation run in computational neuroscience. As it can be seen, *conducting experiment* and *running simulation* are two distinct iterative loops connected by a feedback process. This process uses *interpretation of output results* to design *new experiment setups* and develop *new cybernetic models*.

GENESIS [22] supports the lower loop within the system as shown above, but it leaves a gap for *setting up and executing the simulations* (e.g. setting up model parameter values, different stimuli, storing the parameters and providing execution statistics). A similar gap was identified for other popular simulation engines [215] like BRIAN [87], NEST [82] and NEURON [103], or the most popular functional simulation engine called Nengo [17]). Moreover, each simulator uses its own programming or configuration language, what leads to challenges in porting models from one simulation engine to another and managing them [49]. These problems triggered the idea of creating a more universal simulation pipeline called *Neural Simulation Pipeline* (NSP).

1.11 Thesis Statement

In his thesis, the author claims that numerical simulations must integrate a robust model development methodology, with adequate testing and simulation steering workflows to increase scientific throughput, and improve utilisation of current and next-generation computational infrastructure, available both on-premise and in-cloud. To this end, there is the need to transform the end-to-end computational experiment workflow from one that is non-universal and manual to one that is standardised and automated. To validate the thesis statement, author:

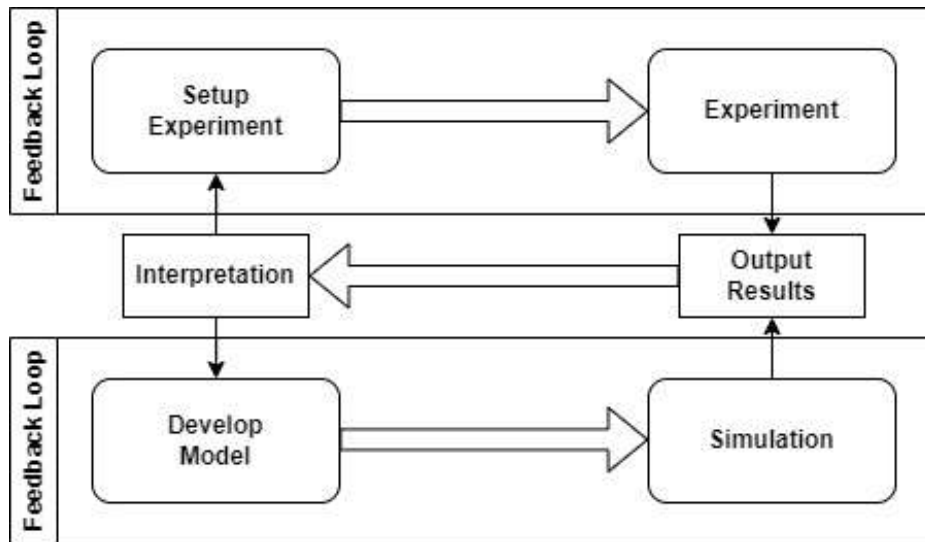


Fig. 1.6 The relationship between experiment and simulation in computational neuroscience. Own figure, based on GENESIS documentation [24].

1. Study the modelling of different spiking neural networks forming Liquid State Machines to understand how the large-scale simulations of liquid models in brain modelling could be improved;
2. Study the impact of different computing resources on model performance, and data generation delivered by GENESIS for increasingly complex neuronal network models;
3. Integrates automation into the model development, testing and deployment process prototypes the neural simulation pipeline using own neurobiological simulations of visual system on GENESIS, with both on-premises and public cloud computing infrastructure.

Author investigates the following research questions:

- How to develop Liquid State Machine models using GENESIS programming language?
 - How to efficiently design and simulate large, realistic neural columns built of Hodgkin–Huxley neurons?
 - How to build a model of visual system using a bio-cybernetic model that would be simple enough to allow for understanding its operations, and at the same time it could illustrate the most important functions of the retina and visual cortex?
- How to automate testing and deployment of such models?
- How to incorporate the prediction of input patterns using machine learning?

- How to best prototype such models?
- What is the simulation cost, how to optimise simulation costs for larger models?
- How to reduce entry barriers to developing and performing large-scale neural simulations for brain modelling?

Chapter 2

Liquid Computing in Brain Modelling

2.1 Introduction

In the second chapter of this PhD dissertation the author describes *what the liquid computing is, and why it is important* for both brain modelling, and as a new computing paradigm. We start with a bit of philosophical discussion on the computer metaphor, comparing the mind to a modern-day digital computer, as well as refer to a wider computational theory of the mind. We then define the concept of Liquid State Machines, highlight their mathematical properties, as well as provide the other examples of liquid models to present the state of art. We follow with the review of large-scale, scientific computer simulation projects using liquid models incl. the famous Blue Brain Project and the Human Brain Project. Finally, the author concludes with a short description of challenges and limitations for these models.

The chapter is organised as follows. This Section 2.1 provides a discussion of this chapter. Section 2.2 focuses on the philosophical discussion of the “computer metaphor” and computational theory of the mind, whereas Section 2.3 explains the concept of Liquid State Machines. Section 2.4 highlights the structural components of Liquid State Machines. Section 2.5 describes the idea and flow of liquid computations in Liquid State Machines. Section 2.6 explains the mathematical properties of Liquid State Machines, whereas Section 2.7 discusses the selected properties of Liquid State Machines. Section 2.8 presents the role of the readout layer. Section 2.9 describes the state of practice and other liquid models. Section 2.10 introduces the key large-scale computers simulation projects of liquid models. Finally, Section 2.11 summarises why the liquid models presented are important, and highlights both their key advantages and disadvantages.

2.2 From Hardware and Software to Brain and Mind

In cognitive science there is a metaphor that “the mind is like a computer” [84]. Once computers arose in early 1940-50s people started looking at them as information processing devices, and a term *neurocomputer* was coined. The brain itself started being widely understood as an information processing device, In these early days this approach was especially influential in American behaviourist psychology, suggesting that one should not talk about ideas that are internal to the mind, because they can not be seen or measured. It was a rigid view, suggesting that if something cannot be seen or measured, it cannot be subject to scientific investigation; so an organism could be treated as a black box device.

The cognitive revolution of mid-20th century was to change this [166], and allow scientists to approach the brain and mind in a different way. If we could say that the mind is “like a computer”, then considering we can write programs and study the behaviour of a computer, in a similar way we could use those programs as metaphors for the operations of the mind. This cognitive revolution allowed us to use a metaphor that “the software is to the hardware, as the mind is to the brain”. In this way we could say that computers run software in a metaphorically similar way to our brain “running the mind”.

McCulloch and Pitts, who published the first mathematical model of a neural network, were also the first to state that neural activity is a digital computation, and that neural computation explains cognition [181].

Advances in computing influenced an increasing number of scientists to believe that the mind itself is a computational system, a position known as *the computational theory of mind* (CTM). Computationalists, so scientists who support CTM, attempted to apply it to certain important mental processes; and the theory itself become a central belief within cognitive science during the 1970s, undergoing further development e.g into functionalism [186].

The idea of functionalism suggests that both mind and brain, similar to the software and hardware, could be understood as systems of rules, states and computational elements. When the software performs the functions via computations on some inputs to produce outputs, running on usually electronic hardware, the brain performs its functions via its biophysical and biochemical operations involving neurons and stimulus responses. How those elements work together as a system is important, but the “physical machinery” in which those systems are implemented doesn’t always matter.

A recent scientific discussion around this theory revolves around the language. Steven Pinker, a psychologist, suggests that the brain’s neural activity functions like a computer, motivating his view by stating that all languages are built on the same universal grammar and that the language mechanism is built into the human brain [182].

Our brain consists of approximately a hundred billion neurons. The data provided by different authors have led to a broad range of 75–125 billion neurons in the whole brain [141]. Neurons are organised in microcircuits, also known as neural columns. They differ by purpose in the human brain [89].

Excitable media, as introduced by Holden [108] in the theory of synchronous concurrent algorithms, provide a great framework for computations. A new approach to microcircuit computing was suggested by Maass [151]. In Maass's Liquid State Machine theory, the brain, or its fragments, are treated as a liquid. The model provides an alternative to the Turing machine [192]. Moreover, a mathematical analysis shows that there are in principle no computational limitations of liquid state machines [151], and that they can simulate any Turing machines [146].

Neural networks are a great tool for modelling different systems, including complex biological or technological systems [203]. Artificial neural networks facilitate biomedical signal processing, biomedical data analysis and interpretation, as well as models for the analysis of system behaviour, including the prognosis of results of selected activities [206].

Several successful applications of the LSM framework have been delivered in the area of artificial neural networks, or to solve engineering tasks such as the design of nonlinear controllers [173]. Moreover, we know that cortical microcircuits seem to be very useful for computing on perturbations [231].

Direct observation of neuronal activity of individual neurons is not possible given the current state of art in human neuroimaging [4]. To achieve brain-scale simulations and to investigate emergent properties of brain circuits we need better building blocks to simulate the whole neural circuits with higher fidelity.

There has been an increased development in the performance and capability of neural simulations in the past years. As a result, simulator and supercomputer technology have now developed to the point where the actual process of setting up and *simulating a realistic neural model is now practical* [98].

In spite of that some key challenges remain. One of them is that the number of synaptic connections in the neural models offering higher fidelity have shown to exceed the current memory capacity of hardware that is available to researchers [138]. The other major challenge to the simulation of neuronal networks is handling both the computations and data generated by the large number of synaptic inputs to a single neuron, and scaling these neurons to the larger structures [125].

This thesis presents a framework for spiking neural network simulation and provides a simple, extensible retina-cortex model. We also leverage a novel simulation setup, and extend the prior work of Wójcik [230], in building a higher fidelity bio-inspired visual system

resembling mammalian visual cortex. The new model has a more complex retina, allowing process of input patterns generated by viewing a resolution of a MNIST size dataset [52].

Author achieves simulation results from different LSM columns. In Chapter 3, the author presents more details about the models, that help in understanding the signal processing phenomena within each LSM column, and how those models are extensible for further research on signal processing in multiple hypercolumns of the brain. Chapter 4 presents a calculation of the Euclidean distance of states between each of the columns of the LSM system to illustrate the differences in spiking patterns.

2.3 Liquid State Machine Concept

The discovery of neuronal microcircuits in the mammalian cerebral cortex provided an inspiration for the *liquid computing theory*. In 2002 Wolfgang Maass and his collaborators published the paper on the computational abilities of such microcircuits [150]. The theory of liquid computing they proposed does not require a task-dependent construction of neural circuits [149]. The computational model they proposed explains how the brain can potentially solve an infinite number of information processing tasks using a network of several simple computational units [153, 150, 149, 148, 152].

The basic principles of Maass's liquid computing theory can be explained by a metaphor of a bucket of water or a lake [210]. Liquid may provide an important computational pre-processing for subsequent linear readouts [147]. In such a metaphor scenario, initially, an ideal, flat and transparent surface of the water is present. At this stage the water has no visible surface deformations. However, at some point, the surface of a lake can become affected by some time-varying disturbances, such as a sudden blow of wind, or by a number of stones thrown into the water. In response, the ripples appear on the surface of the water. Apart from them, the circular swirls appear around the area of where the stones had been thrown into the water. In a short period of time the system arrived to a dynamic, but unstable state. The creators of liquid computing theory intuitively felt that in such circumstances the water surface will be mapping information about the history of its excitation in a form of the *fading memory*, which was formally described in this paper [151].

It is important to note that when the interaction of reservoir with the time-varying disturbances finishes, the surface of the reservoir will again start to calm down, and it will ultimately return to its original flat and transparent state. A theoretically infinite number of ripples may appear as a result of the interaction, evoking a different and unique response each time such an interaction happens. Apart from the lake, other metaphors of liquids are used [210] e.g. buckets, rivers, coffee cups, beer mugs. The abstract concept of *liquid*

computing as a notion of a liquid state, is common in explaining the theory of liquid computing. This state, although unstable, encodes and stores information about a time-varying input and the type of liquid's excitation in the form of a fading memory. Liquid State Machine is also one of the three main computing models for neural microcircuits, with other important models being the Echo State Network (ESN), and the Backpropagation-Decorrelation Model [113].

2.4 Structural Components of Liquid State Machines

Building on the water surface metaphor from Section 2.3, one can imagine a team of experts, who use high quality photographs of the lake's surface to tell why certain disturbances on the water had appeared. In such a scenario, one trained specialist could learn how to estimate the direction from which the wind had blown, while another (also looking at the same photo) could be trained to guess the number and weight of stones that had been thrown into the water. The photo of the lake can therefore be treated as *a record of the liquid state*. The team of experts corresponds to the readout layer in both the LSM model presented by Maass [153, 150, 147] and the ESN model proposed by Jaeger [120]. This element is indispensable for any liquid computations.

The creator of liquid computing theory Wolfgang Maass highlighted that each neuronal microcircuit acts as an independent, real-time computation unit called *Liquid State Machine* [150]. Each LSM is capable of processing a continuous stream of multi-modal input from a rapidly changing environment. As he describes, the framework [151] is not only compatible with biological constraints (e.g. his parameters of neurons and synapses were chosen to fit data from microcircuits in rat somatosensory cortex), but it actually requires these biologically realistic features of neural computation to operate better than Turing machines. The abstract model he proposed was inspired by our brains, where heterogeneous columns of neurons are not task-specific [194]. This means that they may possess potentially universal computational capabilities.

The Liquid State Machine simulated in a computer system is therefore a simplified model of the cerebral cortex [235] in which individual neurons are arranged in hyper-columns. These columns, similar anatomically and physiologically to each other, can process several complex tasks in parallel. The general architecture of LSM is presented in Fig. 2.1. As per definition each Liquid State Machine is built of the three main modules: an *input layer*, at least one *liquid column* or *reservoir*, and the *readout layer*.

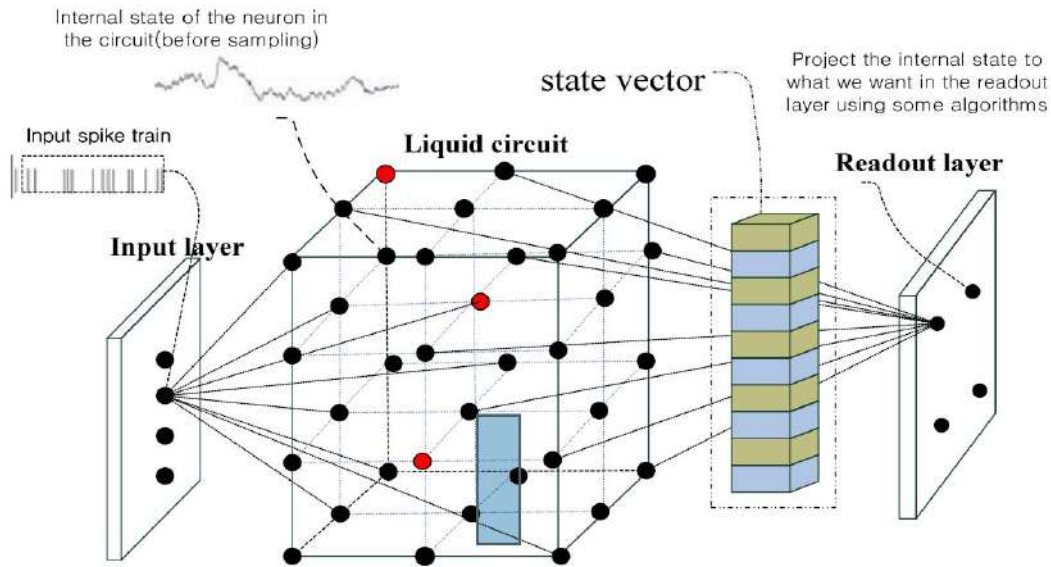


Fig. 2.1 Diagram of Liquid State Machine as presented by Huang [113].

2.5 Liquid Computation with Liquid State Machines

The flow of liquid computations starts with a time-varying input signal $i(t)$ that is fed to an input layer, which is connected to selected neurons of a liquid column. In Maass' original work [153] the input signal was applied to 30% of randomly selected neurons of a column containing 135 simple threshold neurons. Neurons in each column are connected to each other in a unique way. The LSM models assume that for any two cells, the connection weight decreases (often exponentially) using Euclidean distance. To avoid rapid synchronisation of the system, the one-to-many connections model is usually avoided. The system's synapses are formed using a given, selected probability. To maintain biological accuracy the LSM models often assume 80% of *excitatory connections*; so synapses strengthening the signal, and 20% to the *inhibitory connections*, so synapses weakening the signal. Exactly that configuration, build of simple I&F neurons described in Subsection 1.7.2 of this thesis, and using the parameters drawn from the Markram's Lab were used by Maass [152] to present his alternative to traditional (von Neumann's) style of computations.

To be more precise, an *excitatory* synapse is a synapse in which an action potential in a presynaptic neuron increases the probability of an action potential occurring in a postsynaptic cell. In contrast, an *inhibitory* postsynaptic potential is a type of synaptic potential that makes a postsynaptic neuron less likely to generate an action potential [184].

The structure of connections within a neural column in the LSM model is usually defined as a statistical process for any two cells in a given column, so without any external adjustments

needed to set the exact connections' parameters. The statistical distribution of connection lengths within the recurrent circuit is for their separation property, and allows to design different types of liquid circuits [151].

The stimulating signal injected into the liquid column propagates through a defined architecture of connections. A system based on such columns does not rapidly synchronise; rather it is dynamic and unstable. This state resembles the state of a hypothetical liquid, and hence a comparison to the reservoir. There are two negative behaviours which could drastically minimise the accuracy of the Liquid State Machine. It is a pathological synchrony, an infinite positive feedback loop resulting in heavy continuous spiking activity, and over-stratification, expressed in the opposite, inability to propagate the input signal through spiking activity [226].

There are multiple physiological parameters that characterise both the biological neurons themselves (e.g. resistance, capacitance, potential, conductance, physical characteristics of soma like shape and diameter, or axon length), and the connections between them (e.g. different types of synapses, including dynamic synapses). The term liquid computations was formed because a microcircuit resembles a container of a specific liquid. This liquid can be "generated" using different parameters to become a filter for LSM's simple readout learning function. Although there is no consistent method of generating liquids for all problems [176], this approach has successfully been used for example for brain modelling in computational neuroscience [70].

This liquid metaphor is valid mainly because of the type of interactions between different neurons in cortical microcolumns, that act as different liquids. As proven by simulations, the dynamics of activity decrease with the distance in a column. The other similarity of LSM to the cortical structure is the fact that a system composed of the limited number of neurons can assume multiple states, and reflect distance-dependent changes of noise correlations [195].

In spite of that, until now no published research has investigated neuronal columns using approaches known from fluid mechanics. It is possible that such a novel approach could shed a completely new light on the whole concept of LSMs, as different types of reservoirs had already been investigated [134] (e.g. these made of a bucket of water, *Escherichia coli* bacteria, material systems, brain, as well as photonic systems).

The input signal processed by the LSM column is directed to a single, or multiple readout layers. These layers analyse information to detect patterns received to the reservoir, and they perform classification of these patterns. The readout layer can be built of a traditional neural network, or any other method leveraging more traditional machine learning (ML) classification algorithms, usually using a simple learning function [177].

2.6 Mathematical description of Liquid State Machines

Alan Turing showed that one can construct machines that are universal for digital sequential offline computing [192]. In contrast to this common computational model, Maass’s Liquid State Machine model introduced in 2002 [151] does not require information to be stored in stable states of a computational system. In the same year a Liquid Computer term was coined as a novel strategy for real-time computing on time series [173]. Such a computer consists of the four main components:

- Input $i(\cdot)$, a continuous input stream.
- Liquid column, or a filter L^N that maps the input into function $Y^N(t)$:

$$y^N(t) = L^N(i(t)). \quad (2.1)$$

- Memory-less readout map m^N .
- Output $o(t)$:

$$o(t) = m^N(y^N(t)). \quad (2.2)$$

In Maass’s LSM architecture described in “Real-time computing without stable states: A new framework for neural computation based on perturbations” [151] the function of time series $i(\cdot)$ is injected as input into the liquid column L^N , creating at time t the liquid state $y^N(t)$, which is transformed by a memory-less readout map m^N to generate an output $o(t)$.

Such approach to computations is much more suitable for parallel real-time computing on analog input streams. It generated several new models of so called reservoir computing, which are also often called liquid state machine, or LSM [173]. In general, these type of models are created of two parts: the reservoir of a highly recurrent spiking neural network, and a readout function characterised by a simple learning function.

The input always fed into the *reservoir*, often known as *liquid*, which acts as a filter. Then the state of the liquid, called the *state vector*, is used as input for the *readout function*. As a result, the readout function trains on the output of the liquid, with no training happening within the reservoir itself.

This process has been analogised with dropping objects into a container of liquid and subsequently reading the ripples created to classify the objects—this analogy popularised the term Liquid State Machine [147, 176].

2.7 Discussion of Selected Properties of Liquid State Machines

The theory of LSMs requires a few clarifications that author found in “Neurocybernetyka teoretyczna” [210]. Firstly, the liquid column L^N is not subject to a typical learning process that we know from machine learning. The learning process, that can be used in building the LSM based classifier can only take place in the readout layer m^N . The liquid column L^N can therefore be considered special type of fading memory itself.

Secondly, multiple readout layers can be connected to a single LSM. Such layers can for example be specialising in performing different tasks. This means that the LSM exhibits inherent ability to process information in parallel using the same computational unit.

Finally, according to the theoretical analyses conducted on the framework [151], there are no *a priori* limitation to the the computational power of the LSMs for real-time tasks with fading memory. Based on that we can assume that for sufficiently complex calculations requiring large memory capacity, computational units, or columns will need to be containing a larger, adequate number of neurons. In their works, Maass and Markram presented a mathematical proof, showing that LSM has the computational capabilities of a Turing machine (Maass et al. [151]; Maass and Markram [149]; Maass, Natschläger and Markram al. [152]).

One of the most useful properties of LSM is its ability to achieve different states for different input patterns. The distance between states of LSM is called *separation property* (SP). A more detailed, mathematical description of this property is mentioned in Subsection 2.7.1 of this chapter. Unfortunately there are no privileged measures to study the distance between the liquid states. As described by Maass [152] due to the dynamics and non-linearity of the system, there is no simple, homomorphic translation of distances of input states into liquid machine states. This means that even a small change to the liquid column stimulus can result in very different liquid response. The use of different definitions of distance norms provides different ways to analyse machine trajectories in their appropriate multidimensional representations. One of the simplest distances that can be used to describe LSMs is the Euclidean distance between the states of two networks for all time steps performed within the simulation time. Author uses this distance in the Section 4.2. The formal representation of this distance can be written as follows:

$$\| Y_u^N(t) - Y_v^N(t) \|, \quad (2.3)$$

where $Y_u^N(t)$, $Y_v^N(t)$ denote the liquid states at time t for the input streams of u and v .

The Euclidean distance of the column's liquid states (or the states of input neurons) is the distance of two multidimensional vectors, whose coordinates correspond to neurons' change over the time. There are other metrics that manifest the SP of LSMs much better than the usual Euclidean norm, but I will not detail them in this sub-chapter. This aspect of separation property can be useful for developing better classification algorithms, and it is discussed in Chapter 4 of this thesis. For now, I will only mention that the literature (e.g. Wójcik and Ważny [232]) suggest that the Bray-Curtis distance dissimilarity metric may perform better than the traditional Euclidean distance.

Fig. 2.2a, derived from Maass's work [151], presents the dependence of 4 LSM states generated by 4 distinct input streams of varying similarity. In Fig 2.2b, we see why the Bray-Curtis dissimilarity metric may perform better; the calculated distance is expressed in larger values, so the level of dissimilarity for the same input stream is higher.

Maass suggested that SP is only dependant on the parameters of the liquid column. He has also shown that when the dynamic synapses are used to build the column, then the SP increases if compared to the LSMs containing static synapses only. This is in line with common sense. The more internal parameters of LSM there are, the more states it can potentially accommodate; so the use of dynamic synapses allows the machine to adapt to the input signal better.

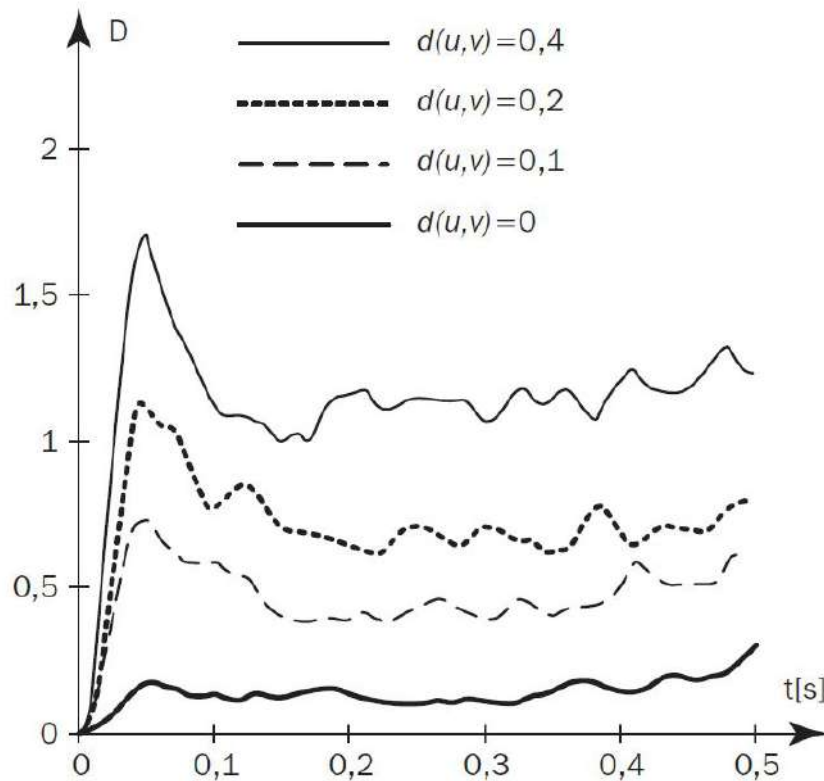
Apart from the *separation property* Maass introduced also the *an approximation property* (AP), described in Subsection 2.7.2, that provides universal computational power regardless of specific structure or implementation. To summarise, SP measures the dispersion between projected liquid states from different classes, whereas the AP indicates the concentration of the liquid states that belong to the same class.

As defined by Maass [151], the universal computational power is provided when the two abstract properties are met: the class of basis filters from which the liquid filters L^N are composed satisfies the point-wise separation property, and the class of functions from which the readout maps n^N are drawn satisfies the approximation property.

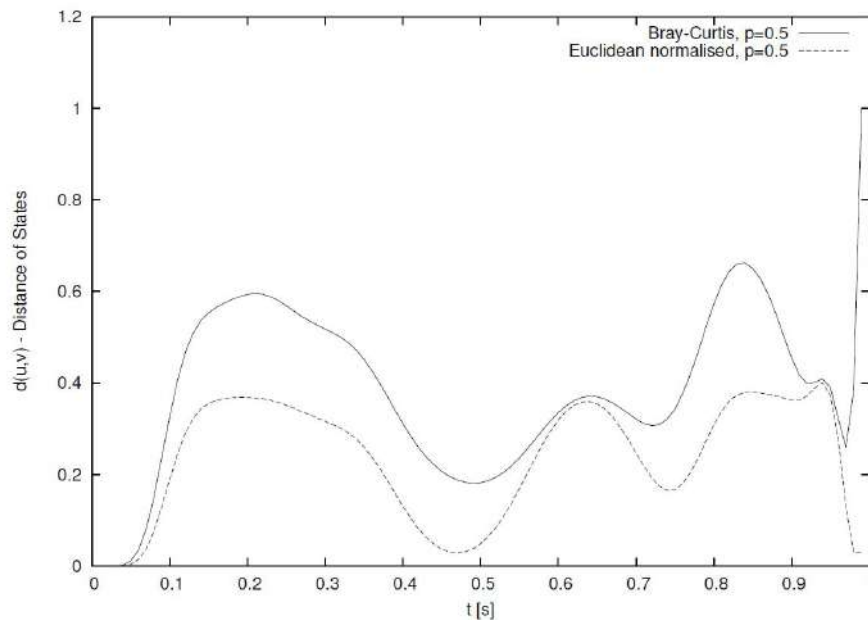
2.7.1 Separation Property

We define that a class CB of filters has the *point-wise separation property* with regard to input functions from U^n if for any two functions $u(\cdot), v(\cdot) \in U^n$ with $u(s) \neq v(s)$ for some $s \leq 0$ there exists filter $B \in CB$ that separates $u(\cdot)$ and $v(\cdot)$, that is, $(Bu)(0) \neq (Bv)(0)$ for any two functions $u(\cdot), v(\cdot) \in U^n$ with $u(s) \neq v(s)$ for some $s \leq 0$. Examples for the classes CB of filters that have this property are the class of all:

- delay filters $u(\cdot) \mapsto u^{t_0}(\cdot)$ for $t_0 \in R$



(a) The state distance increases with the distance $d(u, v)$ between the two input spike trains u and v . Plotted on the y-axis is the average value of Euclidean norm as in Maass [151].



(b) The distance of states $d(u, v)$ calculated in the Euclidean and Bray-Curtis metrics for the probability of interconnections $p = 0.5$ as in [232].

Fig. 2.2 The separation property of Liquid State Machine in Euclidean and Bray-Curtis measures of dissimilarity.

- linear filters, with impulse responses of the form $h(t) = e^{-at}$ with $a > 0$
- non-linear filters, as defined in Neural Systems as Nonlinear Filters [153]

A liquid filter L^N of an LSM N is said to be *composed* of filters from CB if there are finitely many filters B_1, \dots, B_m in CB , to which we refer as basis filters in this context, so that $(L^N u)(t) = (B_1 u)(t), \dots, (B_m u)(t)$ for all $t \in R$ and all input functions $u(\cdot)$ in U^n . In other words, the output of L^N for a particular input u is simply the vector of outputs given by these finitely many basis filters for this input u .

2.7.2 Approximation Property

A class CF of functions has the *approximation property* if for any $m \in N$, any compact (e.g., closed and bounded) set $X \subseteq R_m$, any continuous function $h : X \rightarrow R$, and any given $\rho > 0$, there exists some f in CF so that $|h(x) - f(x)| \leq \rho$, for all $x \in X$. The definition for the case of functions with multidimensional output is analogous.

To summarise, the two theorems proposed by Maass [151] imply that there are no serious a priori limits for the computational power of LSMs on a continuous functions of time, and they provide a theoretical foundation for approximating any biologically relevant computation on discrete spike trains in continuous time by LSMs.

In this approach only the synapses of the readout neurons need to be adapted for a particular computational task, and the current state $x(t)$ of a microcircuit at time t holds all information about preceding inputs.

2.8 The Role of Readout Layer

A critical component of any LSM is *its readout layer*. This layer is needed for use of the separation property, that is “embedded” into liquid column of the system, in order to enable the system to perform classification tasks. Different “readouts” can extract output spike trains. As figuratively illustrated in “Neurocybernetyka teoretyczna” [210] this layer can be compared to an expert standing by a lake, who can determine the nature of the lake’s / liquid’s excitation based on observations of its internal state (e.g. waves on the lake).

The design of readout layer can be performed in three steps:

1. Identification of the neurons in the liquid column that should form the inputs to the readout layer. This selection can range from a small fragment of the network, to all the neurons in a border case.

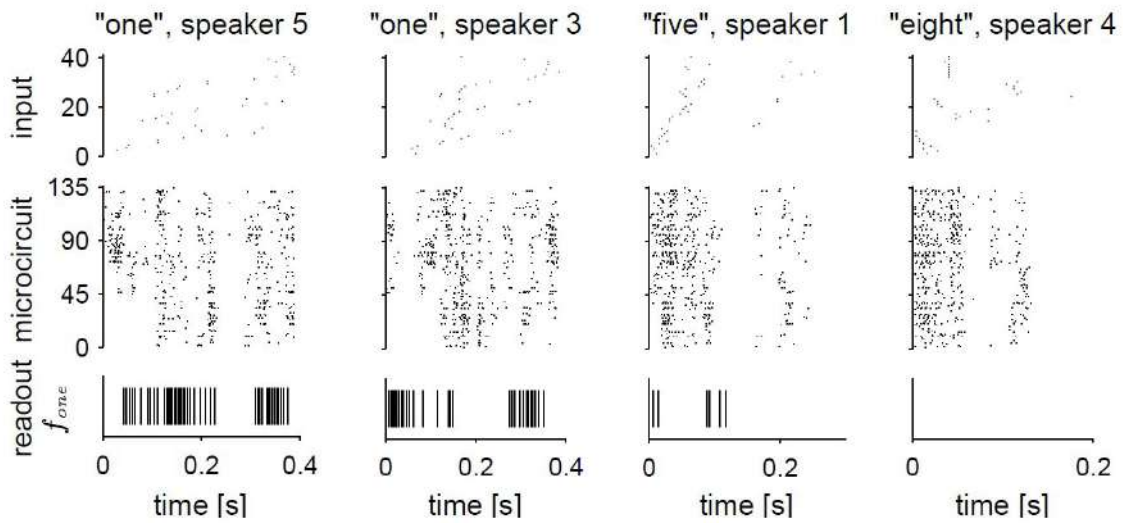


Fig. 2.3 Application of LSM model to the speech recognition, as presented by Maass [150]. Top row: input spike patterns. Second row: spiking response of the 135 I&F neurons in the neural microcircuit model. Third row: output of an I&F neuron that was trained to fire as soon as possible when the word “one” was spoken, and as little as possible else.

2. Recording the time-varying states of the machine $y^N(t)$ for different input patterns $i(t)$.
3. Applying a supervised learning algorithm for the pairs $(y^N(t), o(t))$ in order to train the readout layer in such a way that its response $m^N(y^N(t))$ maximises similarity to the desired output $o(t)$.

The readout layer’s ability to classify correctly the LSM states is called the *approximation ability*. Fig. 2.3 presents state plots of the input, microcircuit, and the readout layer of the LSM system, whose column was built with 135 I&F neurons.

A sample application of this system, as explained by Maass [150] was in the speech recognition task. The LSM was trained to recognise each of the 10 words spoken by 5 different female speakers 10 times. All of the 500 input files had been encoded individually in the form of 40 spike trains for each input, with at most one spike per spike train signalling onset, peak, or offset of activity in a particular frequency band. The network was able to recognise the pattern of “one” with an error of 0.15. The result achieved by this simple LSM system won a well known internet competition in 2000, and sparked interest in LSMs [110].

The top-most part of Fig. 2.3 illustrates that all the four input signals generated by different speakers look alike for an untrained human eye. The spiking pattern of 40 input neurons (represented by small, black points) are indistinguishable for the author of this thesis, who wears glasses. The small points in the middle (‘microcircuit’ layer) indicate

the time-varying activity of the individual neurons in the liquid column built of 135 I&F neurons. Their spikes form visibly more complex patterns in this layer than in the top layer. This is because of the dynamic reaction of the liquid, as well as the fact that there are simply many more active neurons in the microcircuit in comparison to the input layer [150]. As it can be seen on 2.3, the task of distinguishing the middle-layer's patterns is also difficult for an untrained observer. The bottom-layer is different. Even an untrained eye can observe that both patterns of "one", generated by speaker 5 and speaker 3, look "similar". We can also distinguish the input of "eight" from "five" easily, and both from the the original "one" without much effort. It is much easier to distinguish the patterns if a *trained readout layer* is applied to the model.

At the end it is worth mentioning that in spite of using a simplified neuron model, the responses of the neural system proposed by Maass and Markram [150] look 'realistic' if compared to the responses of biological systems. As the authors claim, this allows computer models not only to demonstrate dynamic effects such as synchronisation, oscillations or neural coding, but the readout layer also allows these systems to run *anytime algorithms*, so algorithms providing "provisional answers long before the input pattern has ended". Such behaviour is typical for biological systems.

As described by Wójcik and Garcia-Lazaro [228] the data obtained by recording the real neurons of rat primary somatosensory cortex manifest the same Self-Organizing Criticality phenomena that was obtained with the LSM simulation.

To summarise, the simple LSM system built by Maass [150] using only 135 I&F neurons was able to distinguish between different words spoken by different female speakers, while the other system of Wójcik and Garcia-Lazaro [228] built using 128 thousands I&F neurons was able to realistically model complex phenomena present in biological systems. Obviously, new classification systems or more advanced models of biological systems could be built with LSMs using the capabilities provided by different readout layers.

2.9 Other Reservoir Models and State of Practice

Liquid computing models use sparse and recurrent SNN connections with synaptic delays in networks of spiking neurons "to cast the input to a spatially and temporally higher dimensional space". The main advantage of these models is that they do not require a costly training of their SNN component. Several spike-based liquid frameworks have proven their effectiveness in temporally varying information processing tasks [191].

The LSM concept is under active development. Recently Wijesinghe et al. [226] proposed to use a group of locally connected LSM reservoirs to form an ensemble of liquids. This

approach could simulate higher levels of connectivity in LSMs that are in close proximity, and lower levels of connectivity for these that are spatially further apart.

A good example of how LSM concept influences neuromorphic computing is Spiking Temporal Processing Unit (STPU) [197], a novel neuromorphic processing architecture optimised to implement neural networks and synaptic response functions of arbitrary complexity using the temporal buffers.

Another recent example of extending the original LSM concept was presented by Ivanov and Michmizos [117] in 2021. Authors call their machine Neuron-Astrocyte Liquid State Machine (NALSM), as it addresses the LSM's underperformance through self-organized near-critical dynamics, similar to its biological counterpart, the astrocyte model. Such model integrates neuronal activity and provides global feedback to spike-timing-dependent plasticity (STDP), which self-organizes NALSM dynamics around a critical branching factor that is associated with the edge-of-chaos. Such a LSM achieves a state-of-the-art accuracy without the need for data-specific hand-tuning.

NALSM was reported to achieve a top accuracy of 97.61% on MNIST [52] dataset (97.51% on N-MNIST [178], and 85.84% on F-MNIST [233]). This is significant, because it means that a LSM achieved a classification performance comparable to the traditional, fully-connected multi-layer artificial neural networks trained using the back-propagation algorithm. If performance is the same, the authors [117] say that "LSM could soon reach the performance of other deep learning models, with the added benefits of supporting the robust and energy-efficient neuromorphic computing on the edge". Their view seems to be in line with the current scientific consensus about the next steps in development of deep neural network architectures.

The other, alternative computational models inspired by or related to the liquid computing concept are Echo State Networks and Extreme Learning Machine (ELM).

2.9.1 The Concept of Echo State Networks

Liquid computing has been extended by the concept of Echo State Networks. They are a type of Recurrent Neural Network (RNN) with a sparsely hidden layer that give the architecture a supervised learning principle while being part of the liquid computing framework. The approach, created in 2001 by Herbert Jaeger [120] is sometimes also referred to as the Echo State Machines (ESM). Although the concept of ESN is similar to LSM, it was proposed independently from Maass.

Jaeger himself [122] compares both models in the following way: "these two models had been designed independently, with different application types and different parameter regimes in mind. Theoretical results on LSMs are quite general, and have been formulated

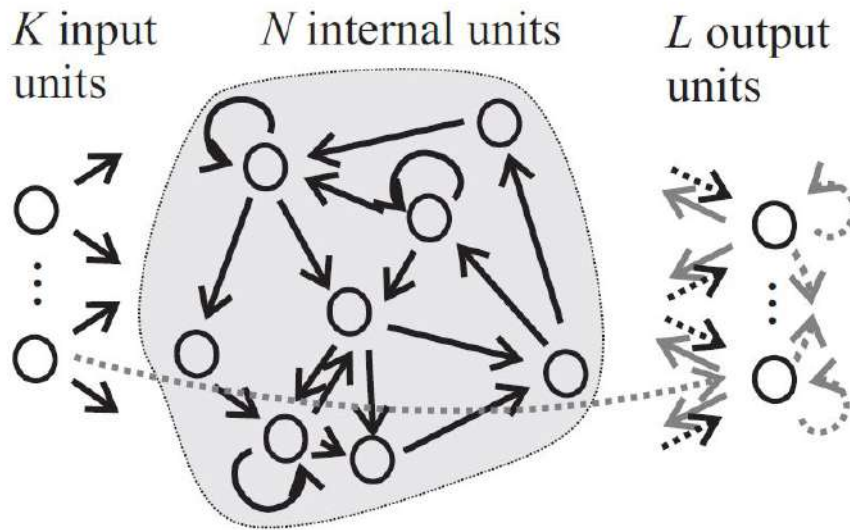


Fig. 2.4 Jaeger’s basic ESN architecture, as presented in [120]; dashed arrows indicate connections that are possible but not required.

within the mathematical frameworks of dynamical system theory and filtering theory. Hence they apply in particular also to ESNs. But since the primary goal of the development of LSMs was to provide a biologically plausible paradigm for computations in generic cortical microcircuits, applications of this model have only been explored for circuits of spiking neurons with a biologically characteristic large amount of internal noise. In contrast, ESNs have been designed to provide high performance for a number of engineering tasks, and have been primarily applied to recurrent artificial neural networks without internal noise, which are better suited for such tasks.”

To summarise, both models compute using a distributed, nonlinear, recurrent neural network with fixed weights (called a reservoir). The adaptation is restricted to the readout layer, what greatly simplifies the training in practical applications.

The two main differences between the models relate to type of neurons used, and their target applications. ESNs use either simple additive neurons with a sigmoid activation function or leaky integrator neurons, whereas Maass considers several different types of neurons [121].

In terms of applications, the ESNs in contrast to the LSMs always use a single-layer neural network as a readout layer. Jaeger’s approach also uses a slightly different naming convention; rather than talking about liquid columns or microcircuits, he talks about *reservoirs of states*. A functional diagram of ESNs is presented in Fig. 2.4

Several engineering applications for ESNs have been presented [121]. They are similar to the applications of other RNNs, so non-generative, and input-driven:

- dynamic pattern recognition,
- time series prediction,
- filtering,
- control.

Jaeger [121] suggests that ESN models for such applications should be set up without the output feedback, and highlights that the stability of ESNs may become problematic when the output feedback is introduced; and output feedback is mandatory for pattern generating ESNs.

As a way of example, an ESN could be used as a tunable frequency generator [210]. In this case, for a given, slowly varying input signals $i(t)$ and the desired output responses of $o(t)$ we get a set of N input-output pairs:

$$N = (i(1), o(1)); (i(2), o(2)); \dots ; (i(n_{max}), o(n_{max})). \quad (2.4)$$

The process is carried out in the three stages:

1. Create a recurrent echo state network containing several hundred neurons; randomly select a number of input and output neurons. Create the return connections from the output (readout) layer to the ESN. The network is called *a reservoir*.
2. Collect n_{max} reservoir states for a given set of input-output pairs $(i(n), o(n))$. We call the internal states of our echo machine $x(n)$.
3. Calculate the weights of the reservoir-readout connections in such a way that the desired response $o(n)$ is obtained from the set of internal states of $x(n)$.

The input signals $i(t)$ induce the desired responses at the output $o(t)$, and the entire system acts as a generator. The machine constructed as per Fig. 2.4 is ready to be used.

2.9.2 Extreme Learning Machines

Another, more recent neurocomputing concept related to Liquid State Machine is called *Extreme Learning Machine (ELM)*. The idea was presented by Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew from Nanyang Technological University (Singapore) in

Algorithm 1 Extreme Learning Machine Algorithm

Require: Given a training set $S = \{(x_i, t_i) | x_i \in R^m, i = 1, \dots, N\}$, activation function $g(x)$, and hidden node number \tilde{N} ,

Ensure:

Step 1: Randomly assign input weight w_i and bias $b_i, i = 1, \dots, \tilde{N}$.

Step 2: Calculate the hidden layer output matrix H

Step 3: Calculate the output weight β .

2004 [111], so at a time, when reservoir computing was still considered a new and fresh approach.

The main inspiration for ELM came from Jaeger's and Maass's [120, 150] realisation that the mathematical requirements for the reservoir computing are (1) high dimensional projection and (2) fading memory. Huang [111] was inspired by the idea that dynamical systems possessing these characteristics "operate at the edge of chaos", and he recognised that this hyper-dimensional projection is a powerful computational tool in itself. In this sense ELMs are similar to the reservoir layers, or liquid columns, but they act without their short term memory component.

ELM are feed-forward neural networks using hidden nodes, whose parameters do not need tuning [111]. In most cases, the output weights of the hidden nodes are learned in a single step, which resembles the process of learning in a linear model [113], and overcomes the slow training speed and over-fitting problems [54]. The definition of ELM, a simple learning method for a single-hidden layer feed-forward neural networks (SLFN), as summarised by Huang [112] is presented in Algorithm Listing 1. H^\dagger is the Moore–Penrose generalised inverse of matrix H , and:

As used by Huang in his works [111, 112], the H is the hidden layer output matrix of the neural network, β is the weight vector connecting the i th hidden node with the output nodes, and T is the training data target matrix.

The relation between them can be written in the following way:

$$H\beta = T, \quad (2.5)$$

where

$$H(w_1, \dots, w_{\tilde{N}}, b_1, \dots, b_{\tilde{N}}, x_1, \dots, x_{\tilde{N}}) = \begin{bmatrix} g(w_1 \cdot x_1 + b_1) & \cdots & g(w_{\tilde{N}} \cdot x_1 + b_{\tilde{N}}) \\ \vdots & \vdots & \vdots \\ g(w_1 \cdot x_N + b_1) & \cdots & g(w_{\tilde{N}} \cdot x_N + b_{\tilde{N}}) \end{bmatrix}_{N \times \tilde{N}} \quad (2.6)$$

with

$$\beta = \begin{bmatrix} \beta_1^T \\ \vdots \\ \beta_{\tilde{N}}^T \end{bmatrix}_{\tilde{N} \times m} \quad \text{and} \quad T = \begin{bmatrix} t_1^T \\ \vdots \\ t_N^T \end{bmatrix}_{N \times m} \quad (2.7)$$

ELM can be applied to different learning algorithms e.g. for classification, regression, sparse coding, grouping, compression, and feature learning. Similarly to the LSM concept, the ELMs have also been studied in relation to biological systems, and that research supports some theoretical claims of ELM.

In 2013 Barak et al. [12] investigated intelligent behaviour. They assumed that such a behaviour requires meaningful integration of several sources of information e.g. to integrate the context with the stimulus, and shape with colour or size. This forces biological neurons to perform both *discrimination*, so responding to similar inputs in a different way, and *generalisation*, so maintaining a consistent response for noisy variations of the same input at the same time. Authors used simulation tools to show that (1) it is the coding level, also known as sparseness of neurons that controls a trade-off between generalisation and discrimination, (2) the optimal coding level depends on the task, but usually the optimal fraction of inputs to which a neuron responds is close to 0.1. To summarise, the intelligent behaviour is driven by the capacity of integrating information, that in biological systems is achieved through mixing sources of information via random connectivity, that form “an easy to read representation of input combinations”. This resembles the approach proposed with LSMs.

Random projections are extremely efficient at generating mixed selectivity and to expand dimensionality, without compromising the ability to generalise. This fact stated by Fusi [76] is important because high-dimensional neural representations are critical not only for learning in ELMs, but also in all recurrent neural networks like ESNs or LSMs; and all these models exhibit a very rich behaviour typical also for biological systems in cortical recordings. This behaviour allows neural systems to solve complex tasks involving short term memory. The alternative for generating high-dimensional neural representations is to introduce layers of non-linear neurons that have random synaptic weights [76].

Finally, new approaches to modelling and approximating neural networks like Closed-form Continuous-time (CfC) could soon completely revolutionise neural simulations, by allowing to simulate brain dynamics composed of billions of neurons and trillions of synapses with biologically realistic mechanisms [94]. This could be achieved by solving the differential equations that describe the interaction of neurons and synapses in a closed-form. These new types of “liquid time-constant network models” are expected to be between one and five orders of magnitude faster than the current models.

2.10 Large Scale Computer Simulations of Liquid Models

In mid-2000s the view that brain (initially it's cerebellum, or little brain) can be considered a liquid state machine was a promising viewpoint for obtaining a better understanding of brain's computational principles. Yamazaki and Tanaka [235] proved that when the neural network structure was reconstructed based on anatomical and physiological data, the non-recurrent dynamics and capability of time representation were reproduced for a little brain. Moreover, such models were robust against Poisson spike input signals, and their robustness could be improved by increasing the number of granule cells. In spite of simulating relatively simple models, they have proven that all the basic dynamic features found in the realistic cerebellar circuit model *were retained* when applying the liquid state machine concept.

Several prominent projects were investigating brain from computational perspective. In author's view the most interesting project focusing on the large scale simulations of the neural microcircuits, so indirectly LSMs, was the *Blue Brain Project* (BPP) [101]. Its coordinator, Henry Markram from École Polytechnique Fédérale de Lausanne (EPFL), is also one of the co-creators of the liquid computing theory [149, 173].

Markram's Blue Brain Project began in July 2005 as a Swiss research initiative at the intersection of neuroscience and Big Data. The aim of the project was to use a supercomputer to ultimately simulate the mammalian (human) brain to deepen our understanding of its functions, and its key disorders [247]. The key achievements of the project until 2006 was a biochemically realistic model of the neural column. The first phase of the of the project was completed in 2007, by simulating a complete neocortical column of a two-week-old rat. The simulated brain structure was based on 10000 biologically realistic neurons with 30 million synaptic connections, all of them embedded in a three-dimensional space.

The simulations in the BBP were run on the IBM BlueGene supercomputers using 8000 microprocessors. The two versions of that machine are presented in Fig. 2.5. As the number of neurons simulated at each stage of the project was relatively modest (a few neurons per core) and the machine was relatively powerful, the system was able to perform all the calculations required for the biologically realistic models in the real time. Higher density of neurons per the single simulation node was planned for the next stage of the project, that was named *Human Brain Project* (HBP) [222], and is discussed later in this section.

The models simulated with IBM BlueGene supercomputer could have been much more complex, and could even consist of millions of neurons. The selection of smaller models was justified by the willingness of running them in real time.

The benefits of the Blue Brain project were expected in medicine, engineering, computer science, and other related fields. The project gave opportunity to verify 100 years of experiments conducted on neuronal tissues using computer simulation. It created a library



(a) IBM BlueGene/L supercomputer used in the early stages of the Blue Brain Project.



(b) IBM BlueGene/Q supercomputer used in the later part of the project.

Fig. 2.5 IBM BlueGene supercomputers from Brain and Mind Institute at EPFL in Switzerland used by the Blue Brain Project [91].

that recorded all the reactions of a simulated neocortical column to all the possible stimulus. The project was primarily intended to provide neurophysiological information for brain scientists and doctors. Project's large computer simulations were reproducing behaviour of a living brain, but it was not a brain with all its cognitive properties.

BBP was therefore not an artificial intelligence project. It was never a goal for BBP to simulate consciousness. As described on the project website at EPFL [247]: "We do not know what is needed for consciousness so no one can really answer the question, let alone whether a digital model can become conscious. One can only speculate. If it turns out that, every atomic interaction in the brain is important to become conscious then one will need to simulate all these interactions to simulate consciousness, which is unlikely to become possible for decades, if not centuries. If, however one only needs to simulate the basic interactions between the neurons then one could begin seeing something that would look like consciousness. But simulating consciousness is not the same as consciousness itself, so it would mostly be useful for us to study the mechanisms that underlie the emergence of consciousness."

Therefore, although the focus on the mechanisms that underline the emergence of consciousness could be an interesting area of study in future, the main focus of the BBP was to answer biomedical and physiological questions; to help us understand the fundamentals of brain operation.

In 2013 the European Union's (EU) Commission awarded the new initiative of Henry Markram a 1 billion Euro grant. As a result of that the BBP has officially been re-branded to HBP, with a much higher ambitions [222]. The goal of this flagship European project with a 10-year time horizon was to improve our understanding of the human brain and to translate neuroscience knowledge into medicine and technology [7].

Apart from attempts to improve the available simulation infrastructure, and in spite of the generous funding, the project contributions were largely related to proving evidence of relationships between neuronal connectivity and functionally relevant brain activity. The BBP ended up with simulating 100 neural columns which consisted of 1 million neurons, and approximately 1 billion of neuronal connections with IBM BlueGene supercomputers. This was roughly the size of a bee brain. The simulations allowed for experimental observation of neural firing patterns, as well as neuronal plasticity [79].

If only looking at the resources spent on HBP, this large project had a limited success. This is because it was not trying to address any particular research questions, or test any specific hypothesis about how the brain works [237].

Nevertheless, one of the findings related to engineering was that balancing the need for the model fidelity and performance becomes increasingly important as model size and

FAR TO GO

The Blue Brain Project has steadily increased the scale of its cortical simulations through the use of cutting-edge supercomputers and ever-increasing memory resources. But the full-scale simulation called for in the proposed Human Brain Project (red) would require resources roughly 100,000 times larger still.

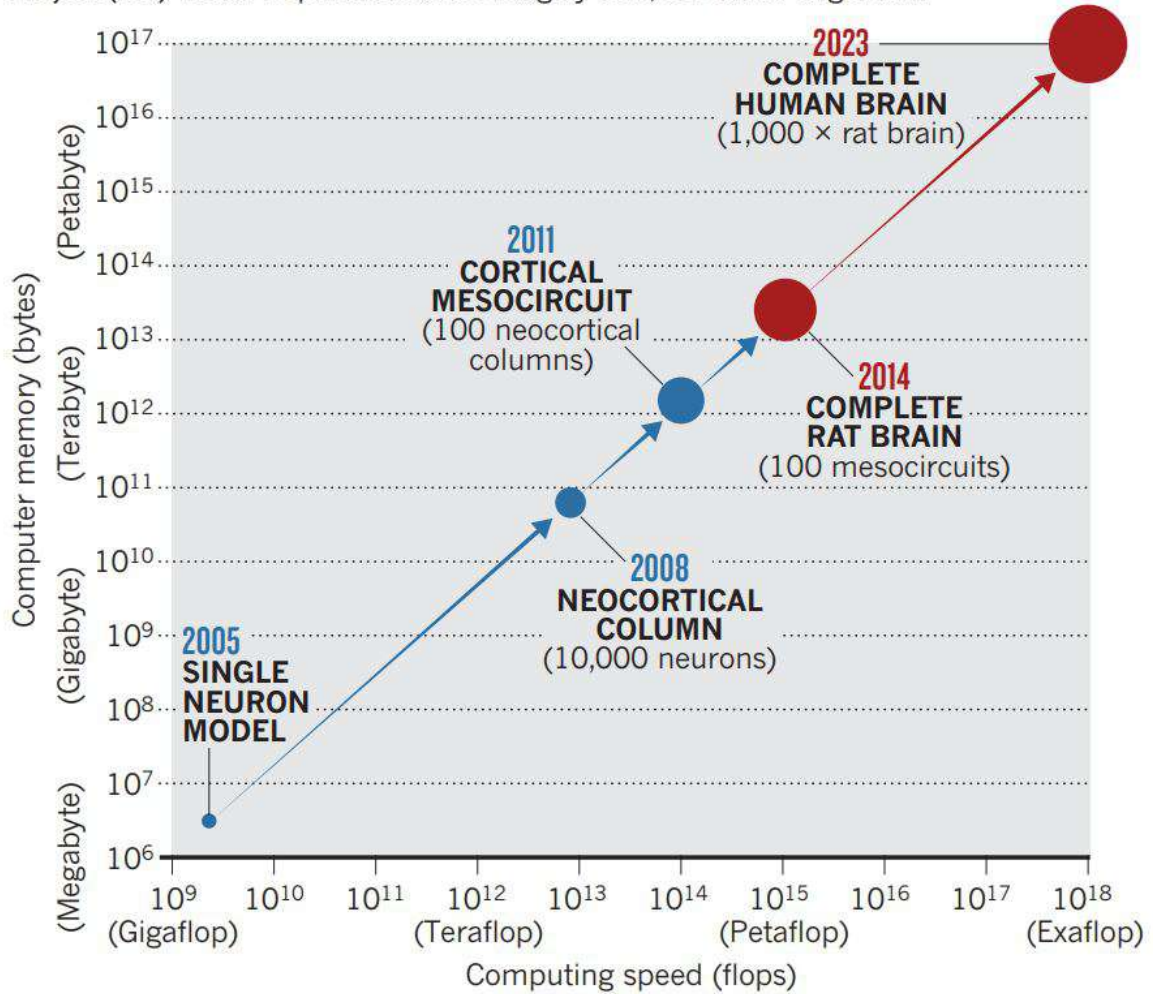


Fig. 2.6 Key Milestones of the Blue Brain Project evolving into the Human Brain Project [222].

complexity grows [247, 91]. To achieve an efficient operation of models built of billions of neurons, we need a clearer understanding of how model features impact computational performance and data generation, and how to automate the the neural simulation pipeline, potentially using multiple simulation engines and compute platforms.

There have been other scientific projects focusing on improving our understanding of how brain microcircuits operate. Brain Research through Advancing Innovative Neurotechnologies (BRAIN) [169] worked on simulating selected, important brain parts. The project used popular commodity computing resources to simulate networks of up-to hundreds of thousands of neurons at the same time, with a goal to understand mammalian brains through reverse engineering.

2.11 Summary

Maass's Liquid State Machine concept is both a new computing paradigm, and a method to simulate large neuronal microcircuits. Mass showed that it is possible to construct recurrent circuits of spiking neurons that can simulate arbitrary Turing machines [146].

It is also the first concept that provided a model-based explanation of the computational abilities of the brain. In 2002 it was used to explain how a continuous stream of multi-modal input from a rapidly changing environment can be processed by stereotypical recurrent circuits of integrate-and-fire neurons in real-time [150]. The model allows to build microcircuits of any complexity with virtually any level of biological realism. Moreover, the simulation results of the large neuronal networks confirm the biological fidelity of such networks, in both the ability of their neurons to oscillate and synchronise [88].

The Maass's generalised approach that defines microcircuits as computational units is an attempt to explain almost unlimited information processing capabilities of our brains. At the same time, it provides a theoretical foundation for explaining operations of our nervous system, as well as researching the process of encoding and processing information [150].

As summarised by Jaeger [120], in the traditional approach when a dynamical system is built using an artificial neural network, the architecture is usually complex as we need to:

1. Train a feed-forward neural network to predict the system output based on the network's input that is drawn from a few delayed instances of the same system's I/O (e.g. based on Takens embedding theorem).
2. Use selected recurrent network architectures for which specialised learning algorithms are known (e.g. a gradient descent on error function), that bring the following challenges:

- (a) Local optima,
 - (b) Slow convergence,
 - (c) Disruption and slowdown of the learning process when the network is driven through bifurcations during learning,
 - (d) Difficulties to learn long-range temporal dependencies due to vanishing (or exploding) gradient problems.
3. Use custom-designed network architecture that overcomes some of the common problems relating to (a-c).

The resulting architectures are often complex, and focus only on solving a single, specific problem. According to Jaeger [120], a good example of such complexity is visible in Long Short Term Memory (LSTM) networks, which have specialised linear memory units with learnable read/write gates to achieve very long memory spans.

The Liquid State Machine, as well as Echo State Network concepts are more universal [120, 153]. They promote a neuromorphic engineering approach [117]. It does not suggest to "engineer" (in a traditional way) any complex neural structures within the reservoir. The liquid of the neural column is general purpose. The time-varying inputs are passed into the column in a natural way, resembling the flow of information observed in real biological systems. This gives computer applications requiring real-time processing of complex input streams (e.g. in robotics) the ability to use simple *tapped delay lines* for storing information about past inputs, and a variety of *adaptive readout devices* to execute multiple tasks in parallel [151]. As a result the same network can be reused to solve problems of different types, and in diverse time scales [56].

Nevertheless, there are also critics of Maass's theory, who have suggested a few areas for improvement of the liquid state machine concept. Konkoli [133] wonders why the exact limits of reservoir computing, and the computing capacity of LSM devices have not been described in literature. He suggests that in fact little is known about exploiting properties of reservoir computing. Although he confirms that the LSM concept is indeed Turing universal (e.g. in the sense of fading memory), that universality is only provided when *the whole class of machines* (understood as the base filters) is considered. He also mentions that there seems to be a general lack of interest in the mathematical foundations of reservoir computing, "which is strange given that these play a prominent role in formulating the concept".

The key disadvantage that is being listed for the LSM concept is the limited ability to steer the machine's internal processes. This makes it almost impossible to explain how the LSM operates on the level of individual neurons, and as researched by Hazan [96], even

small damages to the LSM's neurons reduce the accuracy of LSM training dramatically, even to "essentially random values".

The other criticism is also that while the theory explains how computational units of the brain function, it does not answer the fundamental question of how the brain processes information as a whole; reaching the cerebellum as the largest biological counterpart to the Liquid State Machine [235].

Moreover, the original LSM model as defined by Maass [153, 150] was also experimentally proven to not to be robust to damages, so it cannot always be seen as a universal model for biological computations. The original results published by Maass only showed robustness to noise in the input data. Hazan [96] has shown that specifying certain types of non-random topological constraints, architectures of connectivity like *feed forward with hubs* or *two way power law* can restore LSM's robustness and its ability to maintain pertinent information over time.

Additionally, at present the implementation of LSM requires considerable computing power. There are also only a few LSM models can outperform conventional neural networks in solving real-world classification or regression problems. The challenges with LSMs or ESMs are often attributed to the complex training, the lack of optimal architecture of the LSM (reservoir), and a little research on hyperparameters [117].

Some of the tasks that can be solved using LSMs can still be solved more efficiently with the typical multi-layer artificial neural networks, using the traditional backpropagation of gradients [204]. Regardless of the controversies surrounding the liquid computing concept, the LSM theory has firmly established itself in both neurocybernetics and in the neuromorphic computing [117], as one of the key computing paradigms.

As summarised by Schuman [191], at present (so in 2022) there are several variants of liquid computing, that span from simple reservoir networks for bio-signal processing and prosthetic control applications to using hierarchical layers of liquid state machines interconnected with layers trained in supervised mode for advanced applications in audio or video information processing.

Chapter 3

Liquid State Machine Models

3.1 Introduction

In the third chapter of this dissertation the author focuses on the *Liquid State Machines models* created as part of his PhD research, and *their simulation setting*. The chapter describes cybernetic models of a visual system. We start with a description of why and how to model the biological (neural) systems, and why the the size of such systems is not always their most important quality. We continue with a short discussion of the order of growth in brain simulations. This aspect is important, because such simulations can very easily become computationally intensive e.g. if some larger parts of the brain networks are considered. We follow with the presentation of author's two simple models of visual system. Each of these models consists of a retina (input), and a cortex (a processing column). The first model called RetNet(28x28,4) is equipped with a retina of 28x28 cells, and four identical LSM columns. The second model is called RetNet(8x5,1), and it has a simplified retina (4x8 cells), but a single column of varying size. Finally, the author concludes with a description of the simulation setting.

The chapter is organised as follows. This Section 3.1 provides a discussion of this chapter. Section 3.2 discusses modelling of biological systems, and why although the most interesting results could be produced by networks with a high level of similarity to the real brain structures, many interesting results and applications could also be achieved by much simpler artificial neural networks. Section 3.3 focuses on the key measure of complexity in computer science, that is the order of growth, and its applicability to the brain network simulations. Section 3.4 highlights the RetNet(28x28,4) model built of Hodgkin-Huxley neurons. Section 3.5 describes the RetNet(8x5,1) model, whereas Section 3.6 explains its variation called DeepRetNet(8x5,1), having a varying column size. Finally, Section 3.7 focuses on the simulation setting for all the experiments described in this PhD thesis. The last

subsection is organised into four parts: Subsection 3.7.1 focusing on the initial experimental setup with Google Colab, Subsection 3.7.2 introducing Single Board Computers and Clusters, Subsection 3.7.3 highlighting the on-premise computational resources received from the Future SOC Lab, and Subsection 3.7.4 explaining AWS Cloud Computing Services used in this PhD thesis.

Overall, this chapter presents the biocybernetic modelling, simulation setup, and highlights why they create entry barriers to the simulation of brain networks.

3.2 Modelling of Biological Systems

In complex systems, the simplicity of a model can be treated as one of its key advantages. This is because for such complex systems we are often simply unable to understand "the whole" system easily. As a way of example, the brain is often considered a complex, multi-element, multi-function system. We are still unable to fully understand the brain. Author of this thesis believes that even simplified brain models e.g. these based on liquid columns, but built using high fidelity Hodgkin–Huxley neurons, can give us new insights into understanding the dynamic processes in the brain, as well as potentially new knowledge on how to build neuromorphic systems.

As expressed by Tadeusiewicz [206, 205] the volume or size of the system is not always its most important quality: “if something is proven to work in a small drop, it would also work in a huge ocean”; so studying the behaviour of relatively small networks of neurons using the right architecture, could allow us to discover new knowledge about these complex systems. With reference to Psychology [205], Tadeusiewicz suggests that artificial neural networks could help us explain the brain-based neuro-psychological phenomena like:

- learning – defined as a knowledge building process, that could be observed through its unsupervised form in both humans and machines.
- “artificial dreams” – defined as spontaneous and unexpected processes, automatically emerging from the natural self-learning procedures.

He suggested that the study of these learning processes could be perceived not only as the way leading to the goal (e.g. of building the most efficient classifier), but as a goal itself. It is especially important for science, because normally the authors rarely describe what happens to the neural network during the self-learning process, or when the neural system operates as normal. The most interesting results could be investigated by *networks with a high level of similarity to the real brain structures*. However, even without that, as long as the neural networks are to certain extent similar to the structures operating in the brains, we could still

consider analogies between processes in the brains and in neurocomputers to be able to build more intelligent systems [207]. This is also what sparked author's initial interest in the LSM, and the related neural concepts.

The simulations of neural networks can be used in several aspects of medicine [212]. Apart from facilitating the ambient intelligence of the environment [165], the neural networks can also be used to study the neural basis of disorders such as Alzheimer's disease, or Autism spectrum disorder, as well as to explore the mechanisms underlying brain development and plasticity [57]. The computer simulation can also be used to investigate the effects of drugs and other interventions on brain function, and to identify potential therapeutic targets for the treatment of central nervous system (CNS) disorders, e.g. using different boundary conditions [83].

Alzheimer's disease is a progressive brain disorder that causes problems with memory, thinking, and behaviour. It is the most common cause of 60% to 70% of cases of progressive cognitive impairment in older adults, and it is estimated to affect at least 2.3 million (range, 1.09-4.8 million) people in the United States [45]. The prevalence of Alzheimer's disease increases with age, and it is estimated that approximately 1 in 10 people over the age of 65 and nearly half of those over the age of 85 have the disease [224].

Autism spectrum disorder is a neuro-developmental disorder characterised by difficulties with social interaction and communication, as well as repetitive behaviours and interests [106]. It is estimated to affect 2.3% of children aged 8 years and approximately 2.2% of adults in the United States, with boys being four times more likely to be diagnosed with ASD than girls. The prevalence of ASD has increased significantly over the past few decades, although it is not clear whether this is due to an actual increase in the number of cases or due to the improved detection and diagnosis [140].

There were several attempts to use computational approach to tackle Alzheimer's disease [57, 42], or Autism spectrum disorder (ASD) [60, 59, 55, 58]. This PhD project aims to deliver new tools facilitating the study of brain function through software containerisation based on Docker, and could be used to inform the development of new therapies and interventions in medicine for a variety of CNS disorders, including Alzheimer's disease or Autism spectrum disorder. However, the anticipated advance goes beyond just brain network simulations, and in author's view it also includes computational neuropharmacology [8], as well as an increasingly important computational psychology, based on the "neuron-like" processing principles; complementing its "traditional" computational neuroscience background [179].

Apart from psychology, the other prominent applicability areas of neural networks modelling biological systems are [209, 208, 210, 211]:

- biomedical signal processing (e.g. electrocardiogram signal classifier),

- biomedical data analysis and interpretation (e.g. through medical image analysis),
- therapy results forecasting (e.g. based on a neural model of the disease and patient's characteristics),
- treatment enhancement (e.g. through personalization),
- modelling human or animal behaviour (e.g. in social informatics, through applying big-data to social sciences),
- automating repetitive cognitive tasks (e.g. through an improved human-machine/human-computer interaction),
- modelling gene expression (e.g. in genomics, through mapping gene dependencies and interactions).

3.3 Brain Simulations and Order of Growth

The computer simulated LSM models based on the architecture of rat primary somatosensory cortex gave experimental results that were “in agreement with the dynamics recorded in neurophysiological experiments on real brains” [227]. The validation of simulations happened on electrophysiological data provided by Jose Albert Garcia-Lazaro and his lab at Oxford University [228].

In a general sense, the approach to science based computer simulations is based on the work of the Dutch mathematician Luitzen Egbertus Jan Brouwer. He suggested that one cannot talk about the existence of any mathematical entities until, one provides a way to construct them. As a result, the mathematical modelling becomes the construction of formal descriptions for real entities[32].

We now know that simulating these processes in brains is extremely complex. Even the well established simulation engines like GENESIS [24] often struggle with simulating neural networks larger than 10000 neurons [44], that is required to simulate *a single* cortical column. Reducing the entry barriers to, and the complexity of the large scale simulations seem an important long term goal for the author of this thesis.

One of the key questions that one has to ask prior to start working on larger computational models of brain networks is how hard the problem really is, assuming the current state of computing technology. To assess that it's worth returning to some well known measures of complexity in computer science. The formal definition of complexity in which computer

scientists talk about problems relates to *the order of growth*, and is expressed with Big-O or Big- θ notations [132].

In this approach the complexity of a problem is approximated by the relationship between the size of the input (e.g. data or function), and the measure (e.g. time) that it takes for a given approach, often formally expressed in a form of an algorithm, to terminate. The complexity can be estimated to be an upper bound or exact. Big-O notation gives only an asymptotic upper bound (not an asymptotically tight bound), whereas when the Big- θ notation produces an asymptotically tight bound on the running time. Both *Big-O* and *Big- θ* notations are standard methods of approximating algorithmic complexity in a way that is independent from hardware or operating system software [131].

It means that as the computational problem gets larger, at where we measure the size of the problem (e.g. through some number n), the amount of time that it takes to solve that problem grows as some constant times n , or lower. It means that the problem won't get harder by more than a factor of some number times n (e.g. square for $O(n^2)$, or cube for $O(n^3)$). A problem that is $O(n^3)$ has a larger order of growth than a problem that is $O(n^2)$, and a problem that's $O(n^2)$ is a harder problem, than a problem that's $O(n)$. In the same way, a problem that is $O(n)$ is a harder, than a problem $O(\log n)$, that grows logarithmically.

The computational complexity (e.g. $O(n^2)$) gives us an approximate, but informative notion of how difficult a computational problem is to resolve, as the problem gets larger. This notion is especially important for modern-day simulations, when several computationally expensive neuronal network models are used. Different algorithms, driven by different neuron models might reside behind these large neural network models of today. Smaller networks might form parts of larger networks, that have a complex set of parameters applied to them, to make the behaviour of the simulated neural system resemble its biological counterpart. To sum up, in author's view it's important to understand the approximate complexity of the brain network models, if they are ever to be productionised.

A crucial element of this understanding is to differentiate between problems that are polynomial in their order of growth (meaning they grow as $O(n^2, n^5, n^{100}, \dots)$) to the problems that grow exponentially, so where a problem grows as the exponent (meaning they grow as $O(2^n, 5^n, 100^n)$, for n larger than 1).

A problem that grows exponentially is always computationally more difficult to resolve in a large-scale, than a problem that grows in a polynomial time. We know that the algorithms with logarithmic complexity perform faster than algorithms with polynomial complexity, which are faster than algorithms with exponential complexity. Computer engineers might be tempted to avoid exponential-time problems, because these could only be addressed in a very small version, unfit for a real-world application or industrial scale. This realisation

often leaves the problems of this type for consideration in future, and with the *theoretical interest only*. There is however, a hope to address at least a few of them, thanks to the notion of linking these two groups of problems by *NP-Completeness*.

NP-Complete problems are especially interesting problems [78], because although such problems can be resolved in exponential time, they express properties allowing us to guess an answer in polynomial time, and validate if that answer is correct, *also in a polynomial time*. Nevertheless, if we repeatedly and reliably want to receive all the correct answers, the programme will generate them in exponential time. This is a valuable distinction of computational problems, as some hard computational problems might actually turn out to be realistic to resolve using spiking neural networks [142], as they result to be NP-Complete problems. Author believes, that this approach could be applied practically one day in the simulation of the whole brain, whose different elements could be approximated using different network models or neuron types.

There is also a linkage between machines and brains in the sense that brain is a highly parallel structure. If one, as Maass [152] thinks of the individual neural columns as tiny computational elements, then there are 2–4 million ($2 \times 10^6 - 4 \times 10^6$) cortical columns in the human neocortex working all at once in parallel [218]. This is an idea of a true parallel computation. The computer science could benefit from drawing from concepts on how brain operates through new neuromorphic architectures. The concept of parallel computational columns, structures that resemble computers with multiple processors, or sometimes clusters of multiple computers, all trying to solve a problem simultaneously and potentially even sharing these partial answers. That's often a very good strategy for reducing the time needed to solve a computational problem, or simply learn from temporal data streams [245].

Nevertheless, in theory this approach does not really solve the difficulty of going from a polynomial to an exponential-time problem. Parallelism as a computational strategy can reduce the time needed to solve a problem, but it can't change an exponential-time problem to a polynomial problem [71], so the distinction remains very important.

In the field of realistic brain network modelling, the current requirements of modelling have overgrown the simplicity of the earliest approaches. Author hopes that within his lifetime we achieve a technological level that will allow us to simulate the whole brain. Nevertheless, if looking at the current state of brain modelling, there will be several problems with mapping the appropriate connections and sub-models. This is because different researchers might develop different parts of that large computational model of “a reference brain”, and the simulations of various networks/sub-networks will need to be independently validated and tested before being combined in a single large model.

If such a reference brain can be simulated in *reasonable time*, the next steps could be to simulate the whole, thinking brain in real time, with the following steps being to simulate that brain faster than in real-time.

Author wonders what the implications of such a technical tool would be for both technical and social sciences. Would they create the super-intelligence? The objectives of this thesis, are much less ambitions, but it seems an interesting open question for a consideration in future.

3.4 RetNet(28x28,4)

The main objective of RetNet(28x28,4) is to examine a large, modular artificial neural network of spiking neurons in a simulation of a LSM system processing visual signals on a computational cluster. The secondary goal is to explore the properties of LSM.

We propose a bio-inspired model of a visual system that consists of two main modules. Our approach is in line with the LSM architecture proposed by Maass [151]. There are two main components of our model:

1. Input (Retina, as presented in Fig. 3.1).
2. Liquid (Cortex, built of four identical LSM columns, each as presented in Fig. 3.2).

Similar to the model presented in [230], the new RetNet(28x28,4) Hodgkin-Huxley Liquid State Machine (HHLSM) model uses the same high fidelity multi-compartmental neurons. The soma of each neuron uses biologically similar voltage-activated sodium and potassium channels. Author builds on a well known conductance-based model describing how action potentials in neurons are initiated and propagated in electrical circuit [239]. The GENESIS parameters we used in our simulations are presented in Tab. C.1.

The Retina was built on a 28×28 square-shaped grid and divided into four patches (2×2). Each patch is connected to one of the four HHLSM columns which simulate Lateral Geniculate Nuclei (LGN), and later the ensemble of cortical microcircuits. The retinal cells are only connected to the LSM column through the LGN layer. Each HHLSM consists of 1024 neural cells placed in a cuboid of 8×8×16. The structure of each column in the models is the same (Fig. 3.2). The model contains four columns in total that form the Liquid stimulated by Retina.

There are 90% of excitatory connections established among layers and neurons of each layer and 10% of inhibitory connections. Additionally, Layers L6 of LSM columns are connected with LGNs of other HHLSMs in the same way (i.e. with the synaptic probability of 10%), simulating the corticothalamic feedback. For simplicity, we did not implement

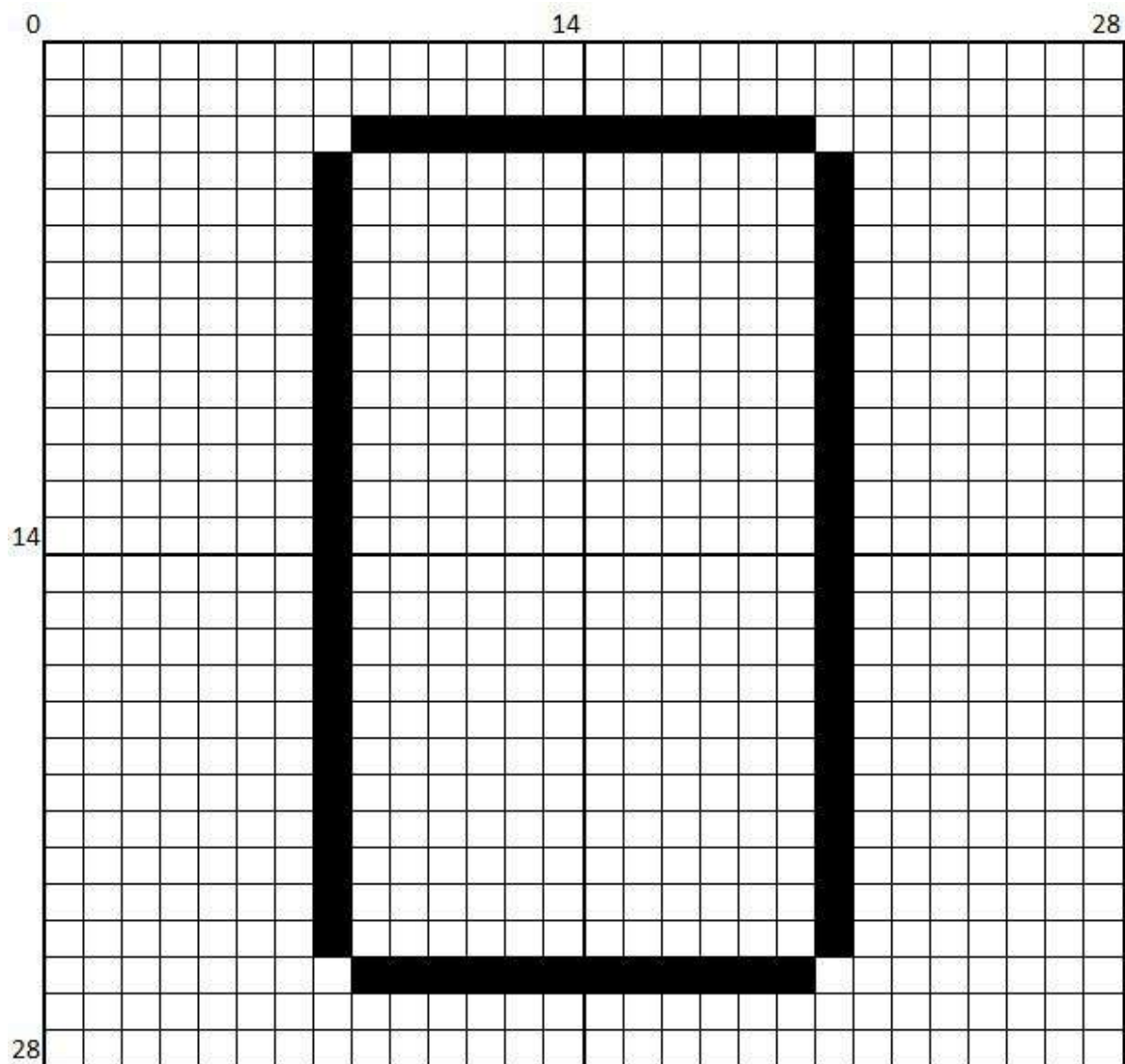


Fig. 3.1 Retina of the proposed visual system with the stimulating patterns (Input).

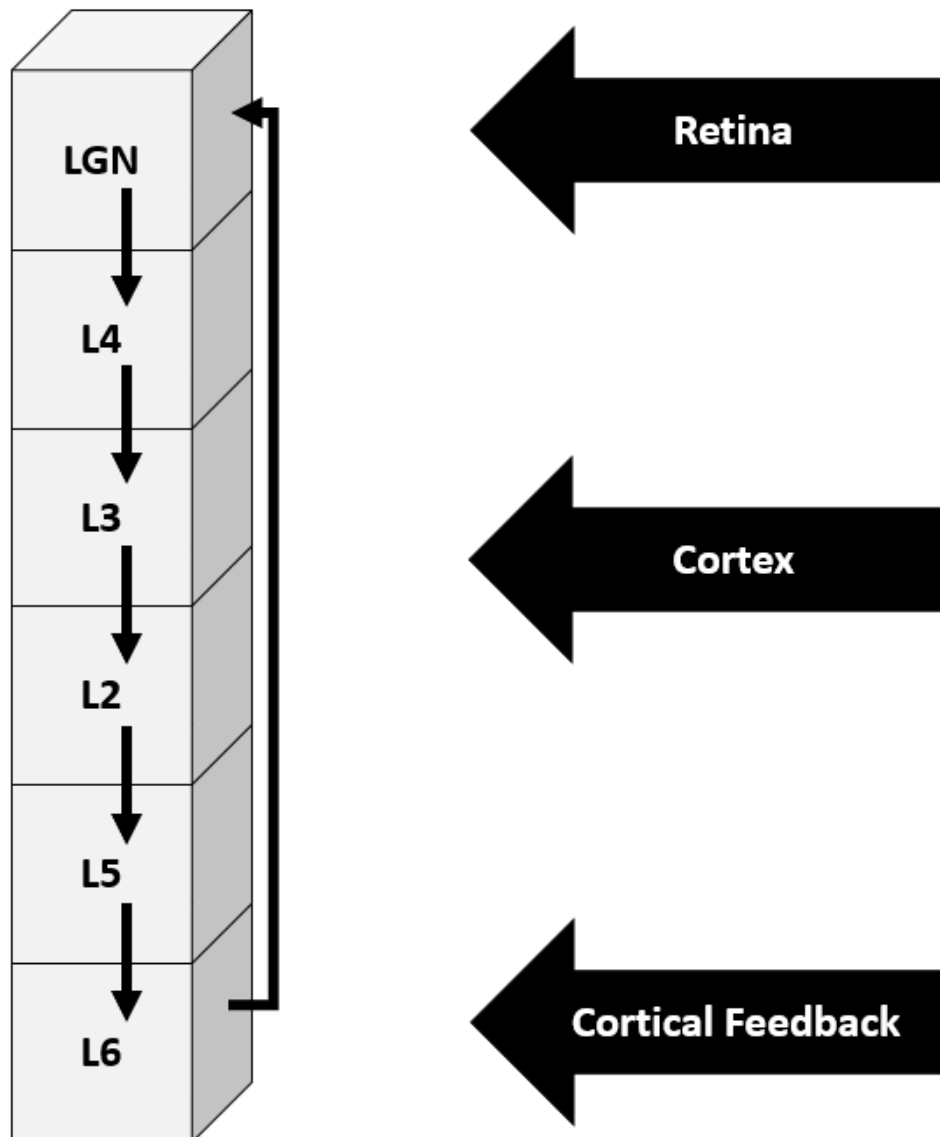


Fig. 3.2 Structure of the LSM column, a fundamental computational microcircuit of our model (Liquid).

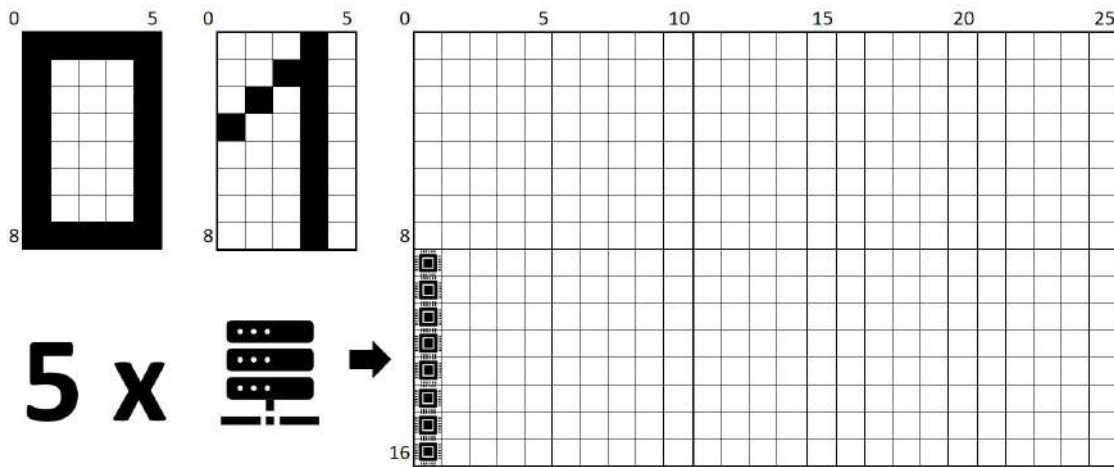


Fig. 3.3 Retina of the proposed visual system with the stimulating patterns (Input).

the pathway of the possible inter-column connections. Each connection in the model is characterised with a delay parameter and random weight. We can treat the Liquid as a single hyper-column made of four independent neural columns, a part of periodic structure of the simulated cortex.

All the simulations for this model discussed were programmed using General NEural Simulation System (GENESIS) [23]. All the neurons used in the simulations were built according to the Hodgkin-Huxley model [107].

3.5 Wide RetNet(8x5,1)

Author proposes a modular, bio-inspired model of a visual system that consists of two main components, an *Input* (acting as a retina of the system) and *Liquid* (acting as a visual cortex, built of a single LSM column). The new RetNet(8x5,1) Hodgkin-Huxley Liquid State Machine (HHLSTM) model uses high fidelity multi-compartmental neurons with voltage-activated sodium and potassium channels. Author has built on a well known conductance-based model describing how action potentials in neurons are initiated and propagated in electrical circuit [239]. The simulation parameters are organised into five groups in Tab. C.1. The exact values for these parameters were achieved experimentally by Yorozu [239] to provide a high biological fidelity of the model. A detailed description of the Hodgkin-Huxley model can be found in [107].

This model is different to prior models e.g. the one presented in by Wójcik in 2006 [230] and the RetNet(28x28,4) presented in Section 3.4. The new model called RetNet(5x8,1) is designed around a single LSM column, as it is intended to become a building block of a

much larger and more complex model that could scale to 1M+ neurons. The final Retina of that target model was built of smaller RetNet(5x8,1) models on a 125x16 rectangle-shaped grid and divided into 5 patches (25x16), with each patch connected to 10 HHLSM columns forming Lateral Geniculate Nuclei (LGN) and the ensemble of cortical microcircuit. A view of this is presented in Fig. 3.3

The retinal cells are only connected to the LSM column through the retina's top LGN layer. Each HHLSM consists of 1024 neural cells placed in a rectangular cuboid of 8x5x25 (1000 HH neural cells). The structure of each column in the models is the same. The target model therefore contains 50 columns forming the Liquids stimulated by Retina(s).

There are 90% of excitatory connections established among layers and neurons of each layer and 10% of inhibitory connections. For simplicity, we did not implement the pathway of the possible inter-column connections. Each connection in the model is characterised with a delay parameter and random weight. We can treat the Liquid as a single hyper-column made of 60 independent neural columns, a part of periodic structure of the simulated cortex.

3.6 Deep RetNet(8x5,1)

The Deep RetNet(8x5,1) model is different to the prior models e.g. the RetNet(28x28,4) presented in Section 3.4 and the Wide RetNet(8x5,1) presented in Section 3.5. This time each HHLSM was built with just a single column in 9 versions using NSP variables described in Chapter 5 Section 5.4.2:

1. 1040 neural cells placed in a rectangular cuboid of 8x5x25.
2. 2040 neural cells placed in a rectangular cuboid of 8x5x50.
3. 3040 neural cells placed in a rectangular cuboid of 8x5x75.
4. 4040 neural cells placed in a rectangular cuboid of 8x5x100.
5. 8040 neural cells placed in a rectangular cuboid of 8x5x200.
6. 12040 neural cells placed in a rectangular cuboid of 8x5x300.

The research on the RetNet(8x5) family of models is important especially for the larger Type 5 and 6 models, as these sizes resemble the real cortical columns, that are cylindrical structures in the brain's cerebral cortex. Although NSP allows for a dynamic change of the size of the RetNet model, this PhD thesis uses 6 standard column sizes in the experiments. In the task simulated, each model's retina receives 3 different stimulation patterns of

“0”, “A”, “1”. This gives us an opportunity to evaluate the LSM system built of an increasing number of neurons in a standardised way.

3.7 Simulation Setting

3.7.1 Initial Setup at Google Colab

In this first approach author proposed to combine the LSM architecture with Apache Spark [242]. Apache Spark can be used as a distributed, fault tolerant data processing engine for extracting insights at scale with near-real time speeds [243]. It is an open source computing framework that unifies streaming, batch, and interactive big data workloads. We use it to improve the analysis of outputs generated by neural columns simulated with GENESIS [23] programming framework.

The fundamental building block of Spark architecture is Resilient Distributed Dataset (RDD). RDDs are in-memory objects on which all the operations on Spark platform are performed, and it happens in a distributed way [241].

An RDD is a collection of entities, similar to rows or records. RDD functionality makes the distributed processing possible by allowing to split the data it contains across all the data nodes of a Spark cluster. RDDs are immutable; once created they cannot be edited, updated, or appended to. They are considered resilient because they tolerate node failures within the cluster, and can be reconstructed in case of a node failure [241, 244].

From the programming perspective they are similar to Java collections, but under the Spark layer they are partitioned and distributed across multiple computing nodes. Unlike in the case of Hadoop and HDFS, Spark RDDs represent the data in-memory for each of the machines in the cluster. The distribution of the data across the computational nodes allows Spark to process the data in parallel. Several processes can be run on an individual subset of data by a cluster [244].

RDDs can be mutated (be appended or changed). There are only two operations that are permitted on an RDD: [243]:

- Transformation (resulting in creation of a new RDD, with all the edits that are needed).
- Action (or request for a result, which causes Spark to execute a set of transformations defined for a dataset).

Spark follows lazy evaluation of operations by keeping a record of the series of transformations requested by the user on the dataset. It groups the transformation in an efficient way when an action is requested. This allows an RDD to be reconstructed even if the node it lives

on crashes. RDDs can be created when a file is read, or a transformation of another RDD is called. Each RDD holds metadata in which it keeps track of where it came from. This feature is called the RDD lineage, and it allows an RDD to reconstruct itself through re-performing all the recorded transformations [241].

Since its beginnings, the platform offers machine learning libraries. Spark 1.x provides support for ML with *spark.mllib*. Spark 2.x, the current version works with *spark.ml* and it offers an entirely new set of APIs for developers, which can work with data frames, and not directly with Resilient Distributed Datasets. The other advantage of the platform for machine learning is that Spark offers libraries for hyper-parameter tuning, allowing us to choose the best model for a given use case [161]. Moreover, the recent developments in the Spark project increased the execution speed on the platform between 10 and 100 times [244].

We performed all the initial simulations described in this paper for RetNet(28x28,4), our first signal processing model, using a simple Spark cluster with a single master node, and two worker nodes. The system was built around computing resources available on a free Google Colaboratory Cloud platform [19]. The machine was equipped with the Intel Xeon processing units running at 2.30 GHz, using 256KB L2 Cache in a single computing component, and having 2 cores per chip. The 64-bit system had 12 GB of RAM, and worked under the control of Linux (Ubuntu 18.04.5 LTS) with the kernel version of 4.19.112+. We installed the latest Spark-3.0.2 with hadoop 2.7 using OpenJDK Runtime Environment 11.0.10. The model was implemented in GENERAL NEural SIMulation System GENESIS v.2.4¹. Both results and source code for the simulations are available on GitHub repository². Fig. 3.4 presents a diagram with simulation setup, including GENESIS and Spark components, along-side their relative roles in the overall approach.

The initial setup allowed us to assess both the components and data generated by the framework. This allowed the author to understand what the difference between *conducting experiments* and *running simulations* is, where the gaps in the current framework are, as well as which of the components are mandatory, and which are interchangeable.

3.7.2 Raspberry Pi and ROCKPro64 Single Board Computers

Exactly a decade ago, scientists from the field of automation and electrical engineering have been excited, praising innovations brought forward at *an increasingly rapid rate*, and opening new realities by introducing the Programmable System on Chip (PSoC) micro-controllers into the engineering education [219]. They said that “engineers’ and educators’ dreams come true of having all their project needs covered in single chip”, because at that time (in

¹GENESIS v.2.4 <https://github.com/dbeeman/genesis-2.4beta-files>.

²LiquidComputer <https://github.com/KarolChlasta/LiquidComputer>.

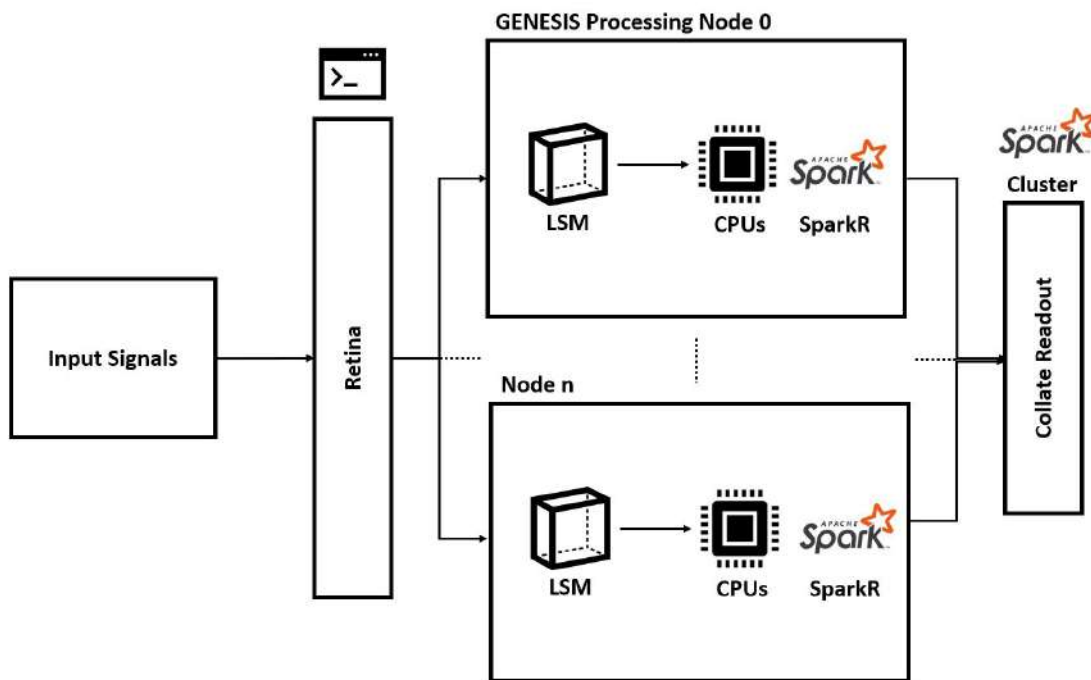


Fig. 3.4 High level simulation setup including GENESIS and Spark components of the framework.

2012) they could use two classes of devices: (1) high-performance PSoC with 32-bit ARM Cortex-M3 microprocessor from Cypress Semiconductors (now Infineon), and (2) ultra-low power RISC mixed-signal microprocessors from Texas Instruments, “an ultimate solution for a wide range of low power and portable applications”.

In the same year, University of Cambridge’s Computer Laboratory together with The Raspberry Pi Foundation released the credit-card size computer called *Raspberry Pi* (RPI). The machine was sold for \$35, with an attempt to “change the world”, by revolutionising computer science education in schools with low-cost and high performance [31]. The original model B was equipped with a powerful Broadcom BCM2835 ARM microprocessor (1 core, 1 thread, and a maximum frequency of 1.0GHz), and up-to 512 MB of RAM. It was also able to run Linux in a graphical environment and provided general purpose I/O (GPIO) connectors for sensors and motors.

Raspberry Pi computing platform was highly successful in several applications: delivering high-quality, low-cost education [234], in higher (engineering) education to build real-time hardware-in-the loop simulators [198], building the scale models for cloud computing infrastructures [217], as wireless sensor nodes [221], or distributed computer vision monitoring systems [214].

The detailed comparison by Maksimovic et al. [154] of RPi and other SBCs have shown that this “ultra-cheap-yet-serviceable computer board is the perfect platform for computing and interfacing with many different devices in a wide range of applications”. Authors suggest that RPi has all the advantages of a PC, but it needs to be provided with the electrical current of only 700 mA. Its performance is better than some popular SBCs development platforms like Udoo (Quad), Arduino, BeagleBone Black or Phidgets “on a general level by computing power, size and overall costs of the solutions” [154].

All the initial experiments in GENESIS were performed using the Raspberry Pi 4 Model B board that is described in Table B.1. The exact configuration of the SBC board was selected and ordered in December 2020, but the computer was only shipped to the author in Feb 2021. This was because the most powerful version of RPi Model B (with 8 GB of RAM) available was available in limited quantity, what increased the waiting time. The board proved very useful for performing neural simulations in the home environment. It was (1) powerful enough to run any Linux distribution with GENESIS, (2) equipped with a 4 core CPU allowing for parallel simulations, and (3) most importantly, the board required very little electrical power (low cost), and (4) could work in the home environment without any active CPU cooling fan, allowing the computer to run simulations 24/7 unnoticeable to the other household members.

When author ordered the additional RPi boards to build a cluster on Jan 15th, 2022, he was soon informed that “the collection of SBC boards had to be postponed until to the end of February 2022”, only to be postponed again to the March 18th, 2022, and subsequently cancelled due to “the lack of availability of the products”.

This is why, in spite of a positive initial assessment of the RPi board, the author of this thesis had to seek for a more available, but equally powerful alternative single board computer. A comparison for various SBCs [127] suggested that their different versions have already been successfully applied and evaluated for education, edge computing and distributed neural network execution.

One of the interesting devices found in the review [127] was ROCKPro64 [115] (RPr). Released in June 2018, it was used by Khaydarova et al. [128] in 2021 to design a cluster of 22 nodes called ROCK-CNN. The machine was used as an execution environment for a locally distributed convolutional neural network applied to object recognition, sentiment analysis and time-series data. This is how the idea for was inspired.

To summarise, the research on bio-cybernetic models described in Chapter 3 of this thesis, whose results are presented in Chapter 4 had been performed under the home conditions using the the *Neural Simulations Cluster* (NSC). The author decided to configure the first RPi board as the master node of the cluster, because this board could provide internet connectivity

through an integrated, dual band (2.4 GHz and 5 GHz) WiFi adaptor out of the box, while the other RPr boards need a special WiFi addition. A more detailed description of the NSC architecture is provided in Chapter 5, Subsection 5.3 of this thesis.

3.7.3 HPI Future SOC Lab HPC Resources

The other computing resource that was used in this PhD project is the Ubuntu 18.04.5 bionic virtual machine (vm-20211005-001.fsoc.hpi.uni-potsdam.de, running Linux 4.15.0-143-generic on x86_64 architecture). It was used as a development workstation, and the on-premise execution environment. Author received access to the machine from Hasso Plattner Institute, allowing him to compile and install all the required software e.g. the latest version of GENESIS simulation environment [24] (version 2.4 from May 2019) and Docker [162] platform (version 20.10.08 from July 2021). The machine was equipped with 8 CPUs: 8 x 2.00 GHz, memory (RAM): 7.79 GB, local disk of 69.38GB.

The second resource used in the experiments described in this thesis was the 1000-Core cluster consisting of 25 Nodes, each 40 CPU cores, 1 TiB RAM (Quanta QSSC-S4R Server System).

Finally, author used some supporting IT infrastructure of the HPI Future SOC Lab. The pipeline was developed using the Docker Registry at HPI at registry.fsoc.hpi.uni-potsdam.de. Some initial exploration has also been done with the Machine Learning capabilities of SAP HANA database management system. [67]. Author managed to connect to that database management system using Python interface, but as per the prior exposure to Spark described in Subsection 3.7.1, limited amounts of data generated when measuring the dynamics of the visual system (spikes), a decision was not to load any data neither to SAP HANA, nor to Spark. That was to avoid building a pipeline using a (proprietary) SAP or (more open) Databricks (Spark) software, with an intention to refocus the efforts on setting up a more universal simulation pipeline.

3.7.4 AWS Cloud Computing Services

Throughout the last three years Amazon Web Services (AWS) has remained the biggest Infrastructure as a Service (IaaS) Public Cloud provider in the world, if measured by both reported revenue and market share. The company achieved a revenue of \$35.4 billion and a market share of 38.9% last year. They were followed by Microsoft, Alibaba, Google and Huawei, collectively amounting to the 80% of the cloud computing market globally last year [80]. These numbers are significant, because AWS is the biggest, while, as the report suggest “cloud-native becomes the primary architecture for any modern computing



Fig. 3.5 Author's Raspberry Pi 4 Model B single-board computer that was used in simulations, and as part of Neural Simulation Cluster.

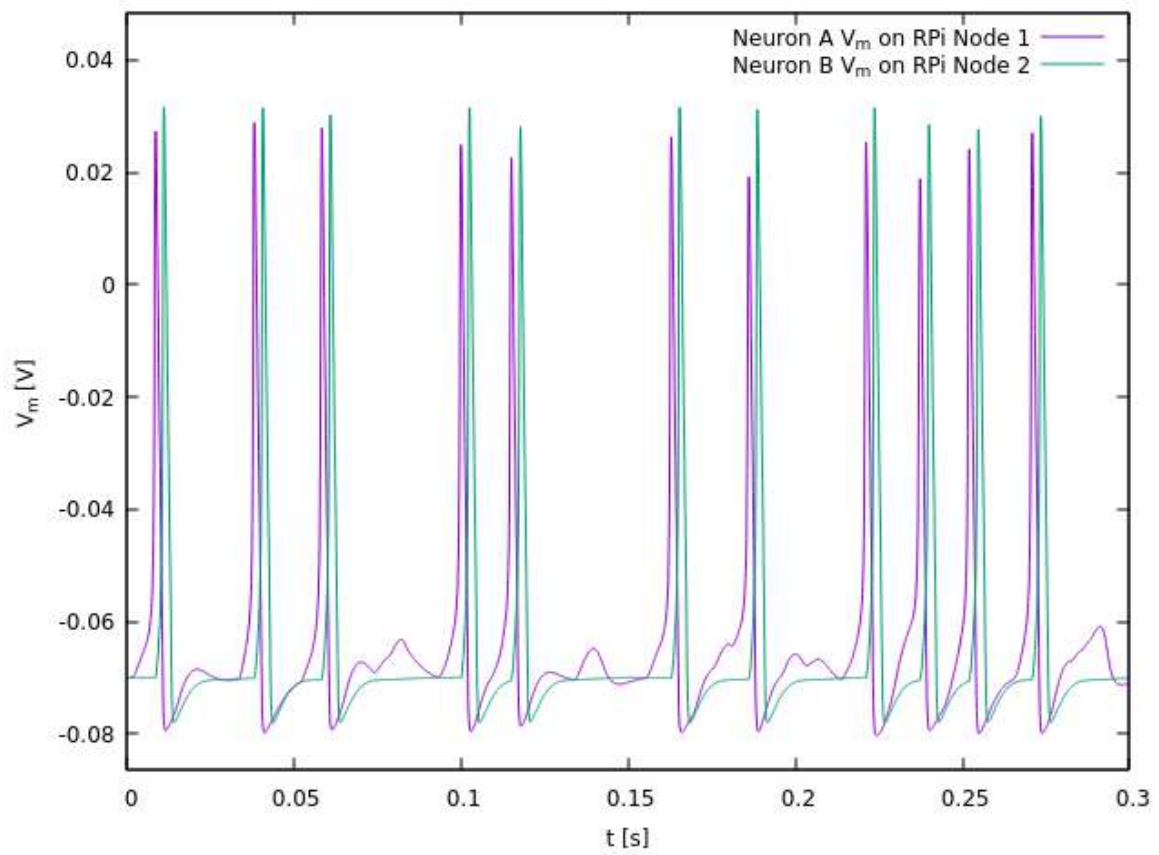


Fig. 3.6 The cell membrane potential in the first 300 ms of the parallel test simulation of two Hodgkin-Huxley neurons, executed on two nodes of Neural Simulation Cluster.

workloads”. In the author’s view this should, and will affect the way large scale computer simulations are performed in future. There might be no return to the large and expensive HPC projects like the Blue Brain Project [155] described in Section 2.10, that simulated a single neural column of 10000 neurons using 8000 cores of the IBM Blue Gene supercomputer (that is 1.25 neuron per core).

The author of this PhD thesis provisioned several services to perform the containerised execution of NSP in AWS, as presented in Figure 3.7. These services are documented on AWS Cloud Products website³. They are: Amazon Elastic Container Service (ECS), Amazon Elastic Compute Cloud (ECC), Amazon Elastic Load Balancing (ELB), whose task definitions were used by Amazon ECS Cluster, AWS Secrets Manager, AWS CodePipeline, AWS CodeBuild, AWS CodeDeploy, Amazon Elastic Container Registry (ECR), Amazon CloudWatch, AWS Simple Cloud Storage (S3), AWS Identity and Access Management (IAM), Amazon Virtual Private Cloud (VPC), and Amazon Route 53 (R53). All these services were used to provision a physical Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz with 14 GB RAM that run each container using a task configured to use 1 CPU ($task_cpu = 1024$), and 8 GB of RAM ($task_memory = 8192$). To summarise, author executed the simulations on Amazon Elastic Compute Cloud (machines), using Amazon Elastic Container Service (Docker), through Amazon Elastic Load Balancing (load balancing) and Amazon Elastic Container Registry (Docker registry) in Amazon Virtual Private Cloud (networking).

All the AWS services were managed through Infrastructure as Code approach [136] with Terraform v1.0.11. As a result, the whole configuration of the NSP cloud environment is stored as Terraform code in the main NSP repository (*nsp-code*), under *infra\core-infra* sub-folder for the VPC configuration, and under *infra\nsp-lb-service* for all the other associated services. This configuration can be used as a reference point for any future deployments of NSP by other members of scientific community.

³AWS Cloud Products website <https://aws.amazon.com/products/>

NSP: Neural Simulations Pipeline

Public Cloud and On-Premise Architectures

KC v2022.11

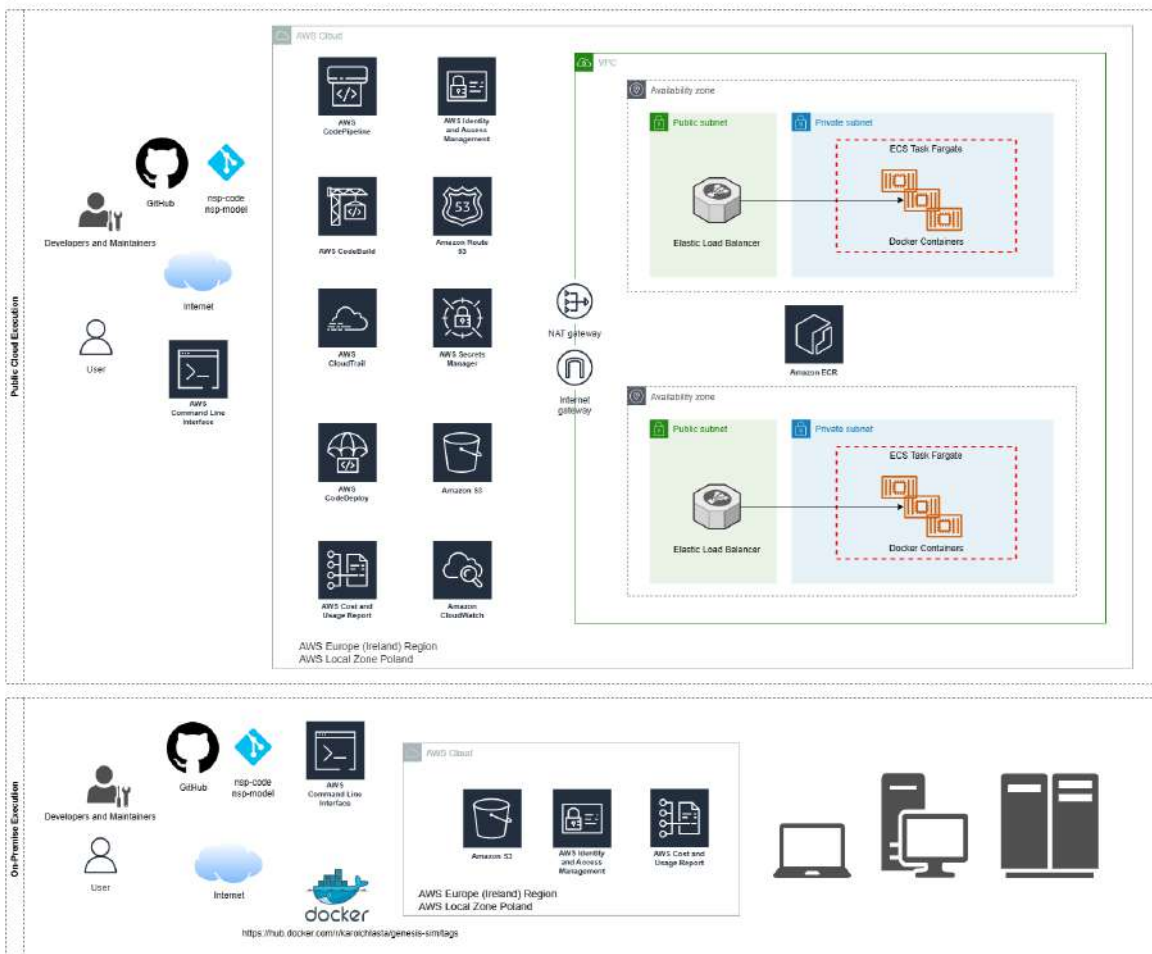


Fig. 3.7 NSP architecture for a full public cloud (AWS) experiment execution (above), and an on-premise execution (below).

Chapter 4

Experimental Work

4.1 Introduction

In the fourth chapter of this PhD dissertation the author describes his experimental work. The chapter presents *the computer simulations based on the cybernetic models and simulation setups introduced in Chapter 3*. We start with a presentation of how biological neuronal networks respond to stimulation with random spike trains of the average rate of 200 Hz, forming a shape of “0” on retinal cells, and how these signals are passed to the subsequent parts of the RetNet(28x28,4) system having four Liquid State Machine columns. We follow with the experiments on a RetNet(4x5,1) system having a variable size of LSM column. We then report on how the LSM readout process has been constructed, what machine learning algorithms were used in the readout’s training, and what evaluation metrics were selected. Finally, author concludes with a description of detailed results for each algorithm evaluated.

The chapter is organised as follows. This Section 4.1 provides a discussion of this chapter. Section 4.2 describes initial LSM experiments with RetNet(28x28,4), whereas Section 4.3 focuses on measuring computational complexity of a single LSM column in RetNet(4x5,1) model. Section 4.4 highlights experiments with RetNet(4x5,1). Section 4.5 describes measurement of Liquid State Machine Approximation Property using RetNet(5x8,1) model. Section 4.5.1 explains how the LSM Readout Process was constructed, whereas its Subsection 4.5.2 provides the overview of twelve machine learning algorithms used in the readout layer’s training, so that our LSM could perform the desired visual task of viewing and differentiating the three patterns of “0”, “A”, “1”. Subsection 4.5.3 presents all the experiment evaluation metrics. Subsection 4.5.4 describes the detailed results of LightGBM algorithm. Subsection 4.5.5 describes the detailed results of Gradient Boosted Trees. Subsection 4.5.6 describes the detailed results of XGBoost algorithm. Subsection 4.5.7 describes the detailed results of Extra Trees algorithm. Subsection 4.5.8 describes the detailed results of Random

Forest algorithm. Subsection 4.5.9 describes the detailed results of Logistic Regression fitted with Max Entropy algorithm. Subsection 4.5.10 describes the detailed results of Logistic Regression fitted with Stochastic Gradient Descent algorithm. Subsection 4.5.11 describes the detailed results of Multi-layer Perceptron algorithm. Subsection 4.5.12 describes the detailed results of LASSO-LARS algorithm. Subsection 4.5.13 describes the detailed results of Support Vector Machines algorithm. Subsection 4.5.14 describes the detailed results of Decision Tree algorithm. Subsection 4.5.15 describes the detailed results of AdaBoost algorithm. Finally, Section 4.6 focuses on the summary of experimental work.

Overall, this chapter presents all the experimental results for the RetNet(4x8,1) and RetNet(28x28,4) in a pattern recognition task. The key contribution is in the analysis of different algorithms used for a supervised training of the Liquid State Machine's readout, as evaluated through all the key classification performance metrics.

4.2 Experiments with RetNet(28x28,4)

As described in Chapter 3 Author created the RetNet(28x28,4) system using 784 cells to programmatically build the *Retina*, and 1024 cells to build each of the LSM's *4 computational columns*. In contrast to the original Maass's LSM using integrate and fire neurons, this architecture was built of a more biologically realistic model of neural cells proposed by Hodgkin-Huxley [107].

The RetNet(28x28,4) visual system was built using 4880 artificial neurons to process input signals through a square-shaped grid of 28x28 pixels. The task in the initial experiment was to stimulate the retinal cells with a pattern resembling the digit of 0. The input signal (Fig. 3.1) was encoded in the Liquid state, and a unique pattern of spikes was observed in each of four LSM columns, as visualised in Fig. 4.3 and 4.4, and measured in Table 4.1.

Fig. 4.1 and 4.2 present two views of the spiking response of our retina grid. Retinal cells chosen for each pattern were stimulated with random spike trains with the average rate of 200 Hz. The spikes marked in black were triggered by input signals simulating the pattern of 'shorter part' of the 0 shape ($2 * 12$ stimulating spike trains), whereas the spikes marked in blue were generated by the 'longer parts' of the 0 shape stimulating our retina model ($2 * 22$ stimulating signals). This is important to observe, because it shows that our retina response generates expected signals when stimulated, and these signals can be passed to the subsequent parts of the system, as presented in Fig. 3.2.

Fig. 4.3 and 4.4 present cumulative spiking activity for each of the LSM columns. We performed a 2D kernel density estimation to visualise the structure of cumulative spikes with contour lines. We observe different dynamics of these spikes generated within each LSM

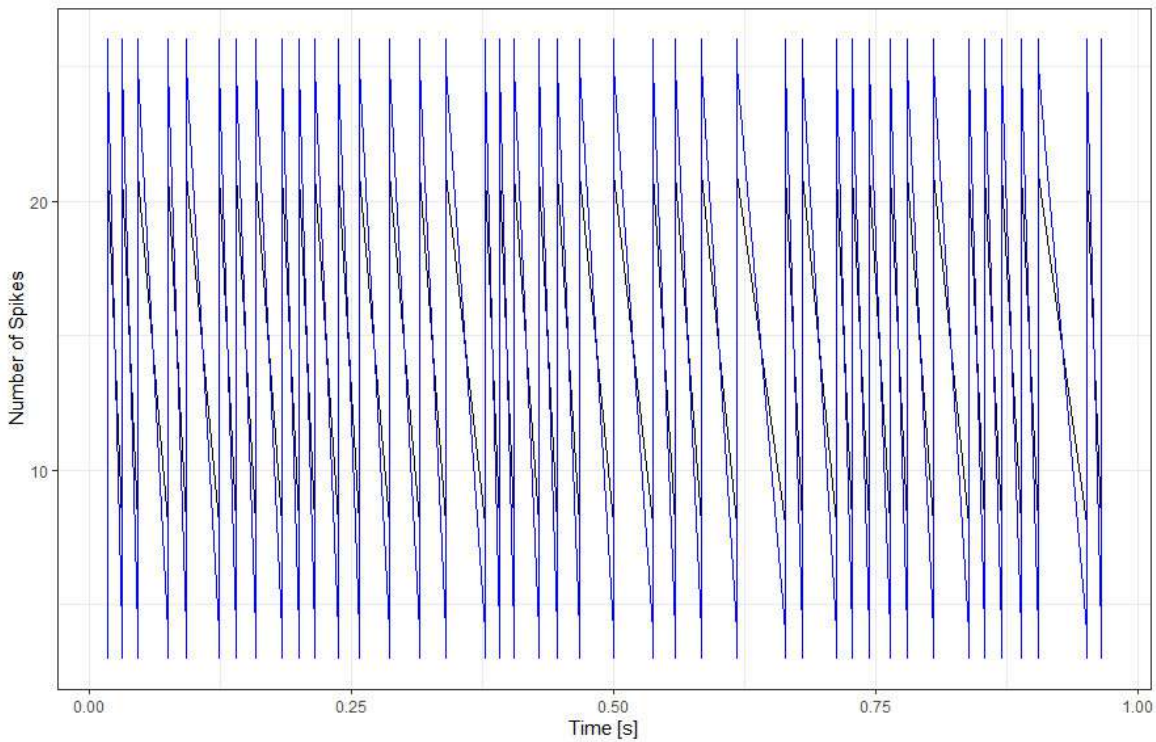


Fig. 4.1 Number of spikes generated by Retina (Input) stimulation for a 2D (28x28) setup within the simulation time of 1 second.

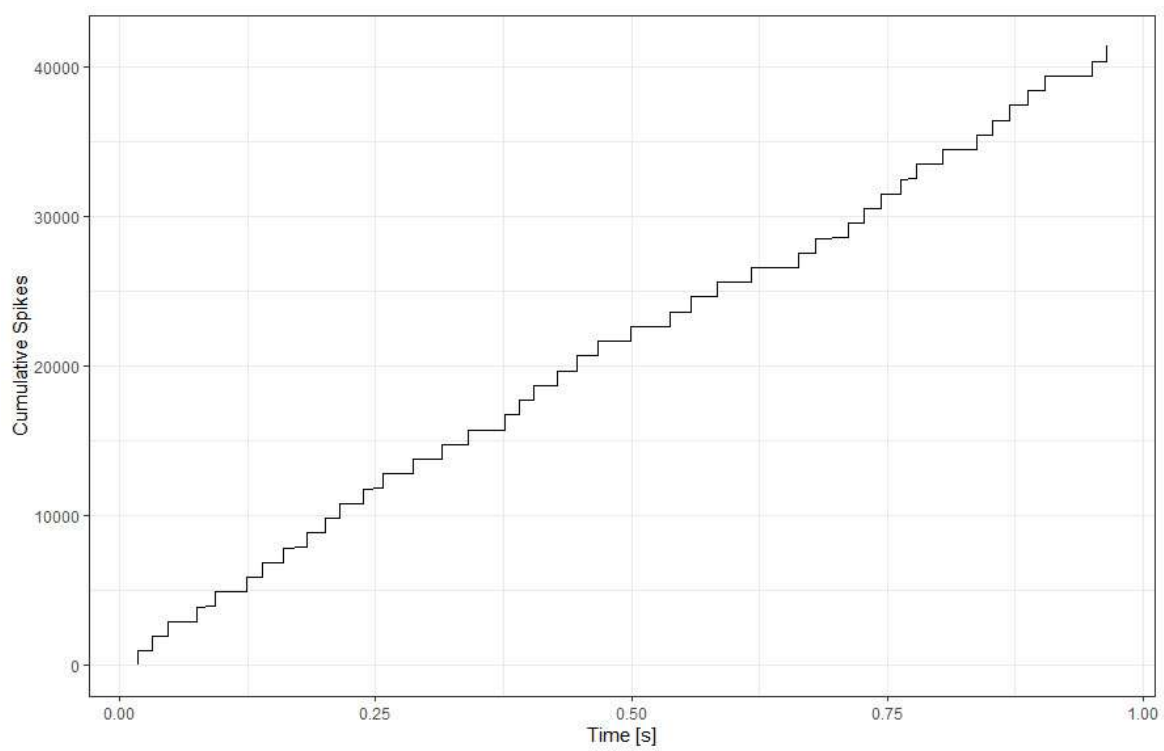


Fig. 4.2 Cumulative number of Retina spikes in each ms of simulation reaching 40000 before the end of simulation time of 1 second.

Table 4.1 Euclidean distance between vectors of states of all four LSM columns in Ret-Net(28x28,4).

Euclidean Distance	Column 1	Column 2	Column 3	Column 4
Column 1	0	3.822376	4.308967	4.375152
Column 2	3.822376	0	1.556321	7.617872
Column 3	4.308967	1.556321	0	8.062106
Column 4	4.375152	7.617872	8.062106	0

column. Such behaviour is typical for LSMs, and can be measured for each column, using their vectors of states.

Finally, Table 4.1 presents the matrix of euclidean distances between four vectors of states of each LSM column during the time of experiment. This different pattern is an important feature for the downstream processing of visual information because we confirm the liquid computing abilities of neural microcircuits. We observe that the spiking pattern of each of the four columns is slightly different, what is numerically expressed in the distance values calculated for each pair of columns.

To summarise, the initial experiment simulated a single second of this simple visual system in 20000 steps using 4 LSM columns. We measured that using the initial setup of the system based on Google Colaboratory Cloud platform, and the simulation time step of 0.00005 second, a single second of the simulation required 285 CPU seconds (4:45 min). Each LMS column generated on average 1.34 MB of spiking data, with retina adding additional 113 KB. Apart from gathering the spiking times for each neuron, our system can also measure cell membrane potential for each neural cell in the simulation. In this approach, the amount of data generated increases approximately by 165 times, so the additional stream generated in this way reaches 1 GB per second. The author has not noticed any unexpected changes of the data stream over the time of the initial runs. The system was stable, able to export output data into the file system, one file per retina and each LSM column.

4.3 Computational Complexity of Single LSM Column

This section attempts to explore the complexity of a single LSM column. At the outset, the author wants to estimate how complex the computational complexity of a single LSM column is. This is important because the human brain's neocortex consists of an estimated number of 2 to 4 million cortical columns, all working in parallel [218].

As per the discussion in Chapter 3.3, the computational complexity of a problem in computer science is defined by the relationship of how the time needed to obtain a solution

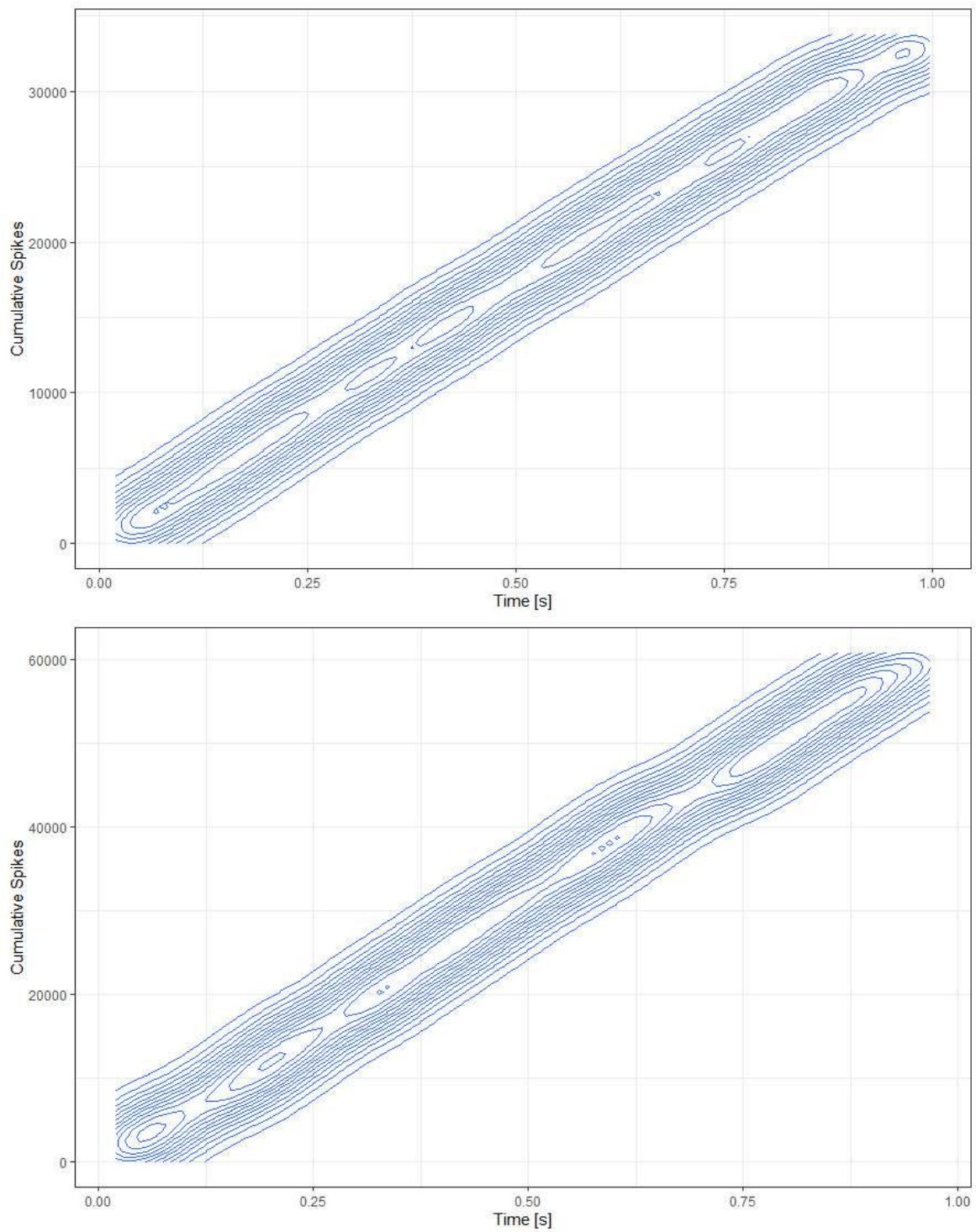


Fig. 4.3 Visual signal processing in the first and second LSM column of the RetNet(28x28,4) system.

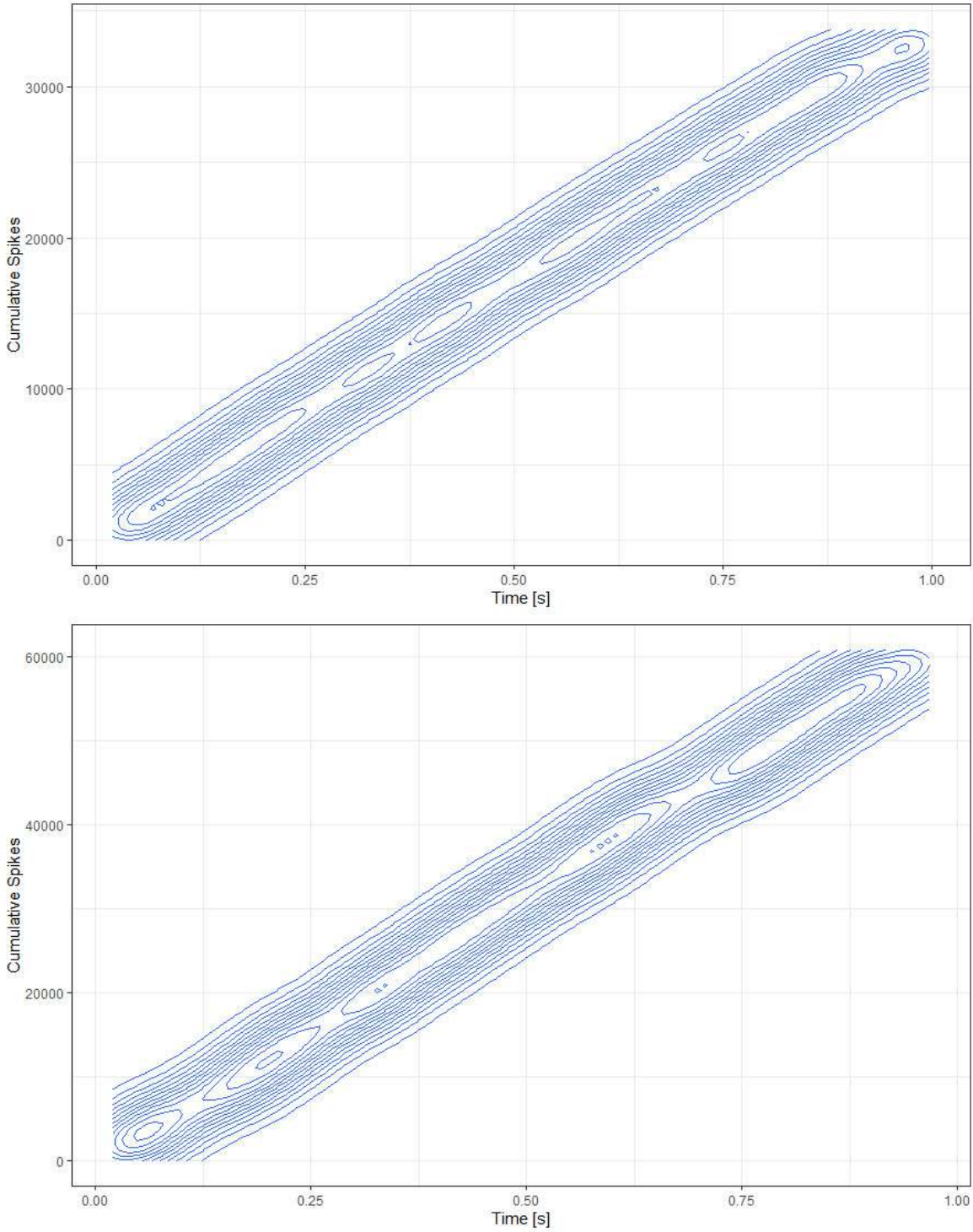


Fig. 4.4 Visual signal processing in the third and fourth LSM column of the RetNet(28x28,4).

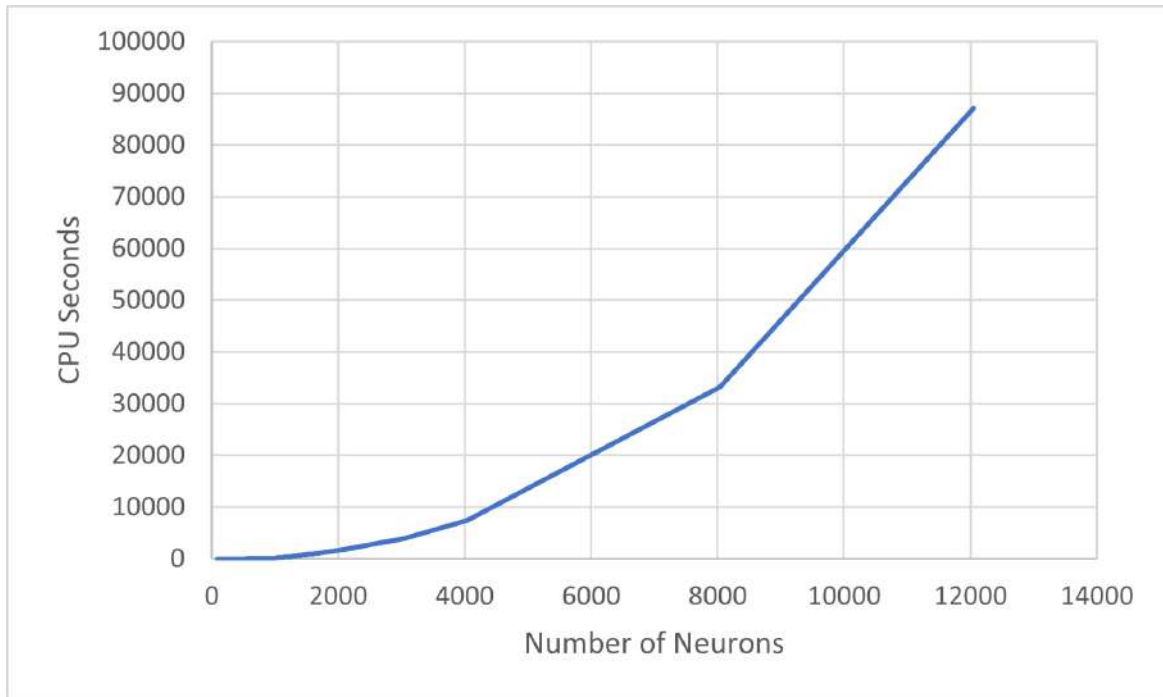


Fig. 4.5 Computational complexity measured in CPU time consumed by RetNet(8x5,1) model simulated with a gradually increasing number of HH neurons ranging from 84 to 12044.

changes with the growth of the problem. In our case it is the CPU time, time spent on executing the simulation code, without waiting time needed for its turn on the CPU, so “real” net time required to simulate a single second of a single neural column; measured 58 times as the complexity of the computational column grew, the column was simulated with a growing number of HH neurons ranging from 84 to 12044, but stimulated with the same pattern of “0”. This relationship is presented in Fig. 4.6, in terms of the spiking activity of such a growing column; and in Fig. 4.5 in terms of its growing simulation time (CPU time).

Both measures, the CPU seconds and the number of spikes have been generated by the RetNet(8x5,1) system using the tools presented in Chapter 5. This allowed for an easy parametrisation of the computer simulation, and its multiple execution using the proposed queuing mechanism. As a result the parabola visible in Fig. 4.5 was estimated using the curve fitting method [93], and the author was able to approximate that the simulation of our single LSM column is a *polynomial time problem*, that grows $O(0.000679468 * n^2 - 1.09334 * x + 716.782)$; so the problem won’t get harder by more than a factor of a number times “n square”.

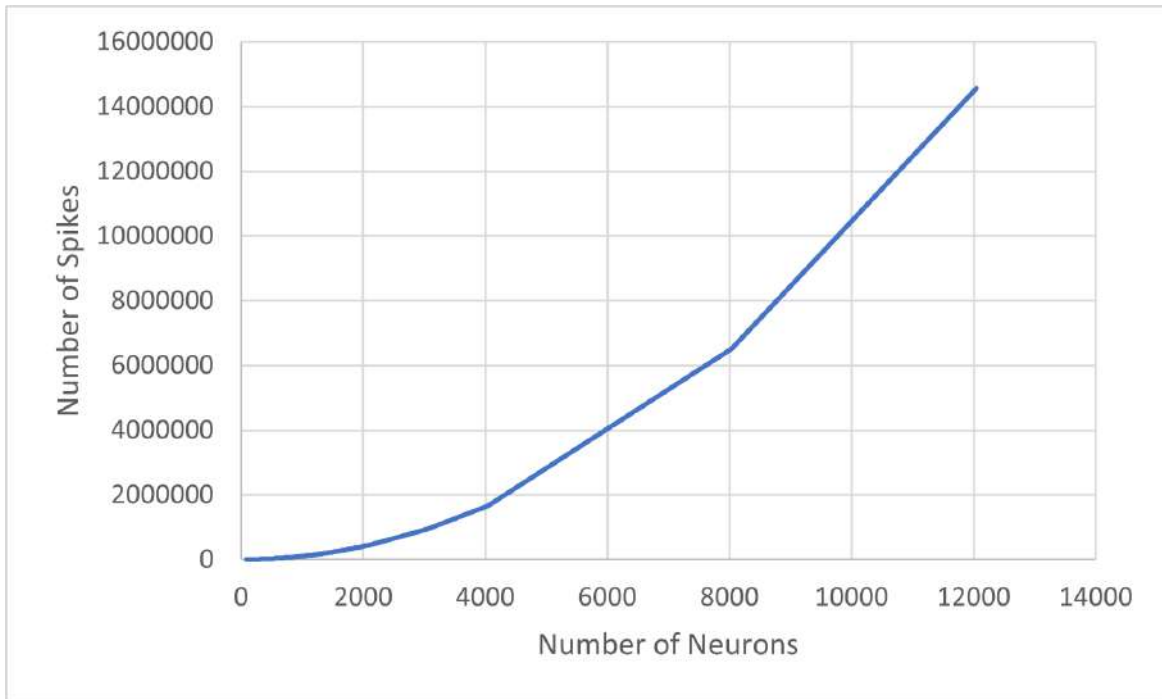


Fig. 4.6 Total number of spikes in RetNet(8x5,1) model simulated with a gradually increasing number of HH neurons ranging from 84 to 12044.

4.4 Experiments with RetNet(8x5,1)

4.4.1 Wide RetNet(8x5,1)

Author built and ran the initial building blocks for the larger LSM models. Initially the small RetNet(5x8,1) blocks were tested using 2 LSM columns in total. These columns can execute in parallel, without using the designated CPU steering commands that are provided by GENESIS ("@1, @2").

A baseline model RetNet(28x28,4) takes about 8 mins 18 seconds to execute on author's own NSC (Raspberry Pi 4B) introduced in Section 5.3. Simulation of a single second of the RetNet(5x8,1) block takes on average 1 min and 29 sec with HPI's VM, and 2 mins 3 seconds on own NSC (Raspberry Pi 4B).

HPI's Future SOC VM running on the HPI cluster is on average only 37.79% faster than the own NSC (Raspberry Pi 4B). The detailed results are presented in Table 4.2.

This initial experiment measured that for a relatively simple base model containing a single LSM hyper-column - RetNet(8x5,1), the difference in execution time is low, even though it was built with multi-compartmental HH neurons (each of the cells contained 2 compartments and 3 channels). An additional simulation time of 34 seconds on Raspberry Pi

Table 4.2 Summary of RetNet(8x5,1) and RetNet(28x28,4) simulations (1 second of LSM system, 20000 steps). Execution in CPU seconds, memory in MBytes, HPI vs NSC.

Computer	Model	Pattern	Neurons	Execution	Memory	Spikes
HPI	RetNet(8x5,1)	0	1040	70.25 s	3.48 (19.07) MB	20828
		1	1040	71.23 s	3.48 (19.97) MB	20814
NSC	RetNet(8x5,1)	0	1040	96.63 s	3.48 (679.81) MB	20822
		1	1040	96.50 s	3.48 (903.52) MB	20820
HPI	RetNet(28x28,4)	0	4880	345.09 s	16.42 (458.66) MB	98838
		1	4880	308.67 s	16.42 (531.45) MB	98821
NSC	RetNet(28x28,4)	0	4880	474.68 s	16.42 (549.70) MB	98829
		1	4880	418.18 s	16.42 (499.99) MB	98806

(vs HPC cluster) seems negligible in an educational or research setting, if one just needs to prototype a model.

Author built and ran 15 RetNet(8x5,1) blocks consisting of 15 LSM columns and 15600 biologically realistic, spiking HH neurons. The results show that the LSM model react differently to 15 different input patterns that were numbers (“0” to “9”) and letters (“P”, “J”, “A”, “T”, “K”). 225 simulations was executed using the Type 1 model. Author measured the CPU Time and memory consumption for the models, as well as validated the simulation setup. The aggregated results of the simulations are presented in Fig. 4.7, Fig. 4.8, and Fig. 4.9. The final version of the RetNet(8x5,1) Type 1 model was evaluated using 4 versions of the liquid column:

1. 1040 neural cells placed in a rectangular cuboid of 8x5x25.
2. 2040 neural cells placed in a rectangular cuboid of 8x5x50.
3. 3040 neural cells placed in a rectangular cuboid of 8x5x75.
4. 4040 neural cells placed in a rectangular cuboid of 8x5x100.

The structure of synaptic connections in each column described above is the same. As the column can have 4 sizes, assuming 15 stimulating patterns connected at the same time, it is possible to evaluate the LSM system built of progressively 15600, 30600, 45600 or 60600 neurons.

4.4.2 Deep RetNet(8x5,1)

That version of the bio-inspired RetNet(8x5,1) model, built using 4880 Hodgkin-Huxley neurons with two main components; an *Input* (acting as a retina of the system) and *Liquid*

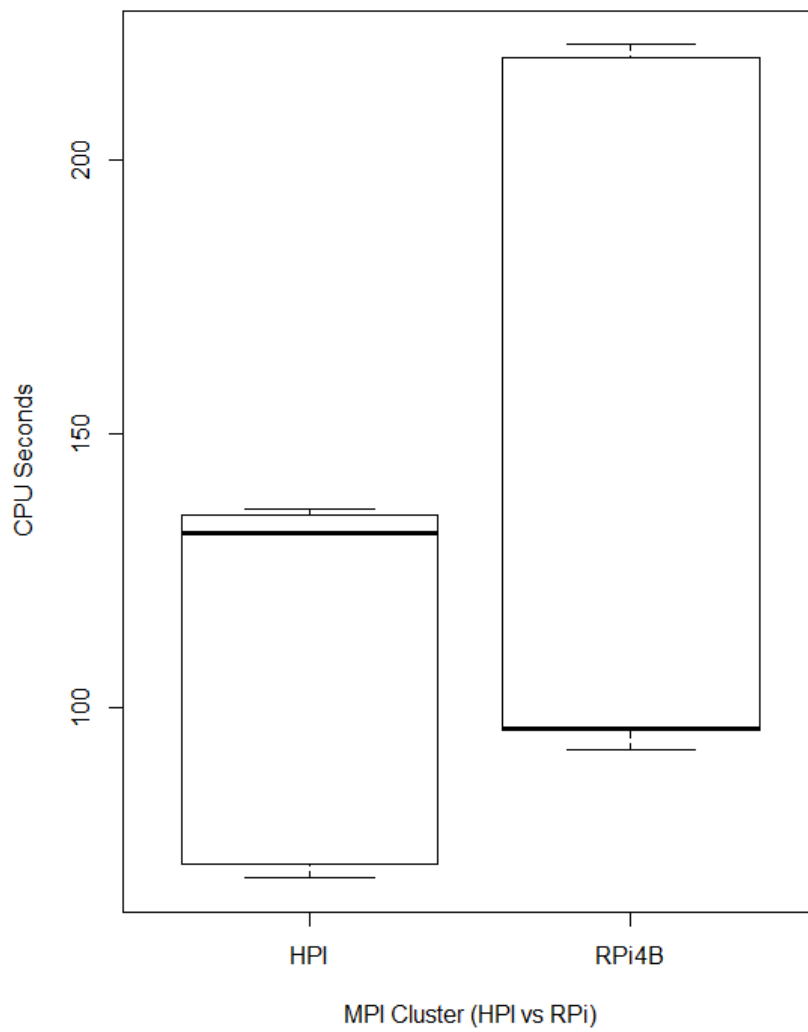


Fig. 4.7 The box plot presenting RetNet(8x5,1) simulation time at HPI vs NSC (RPi4B).

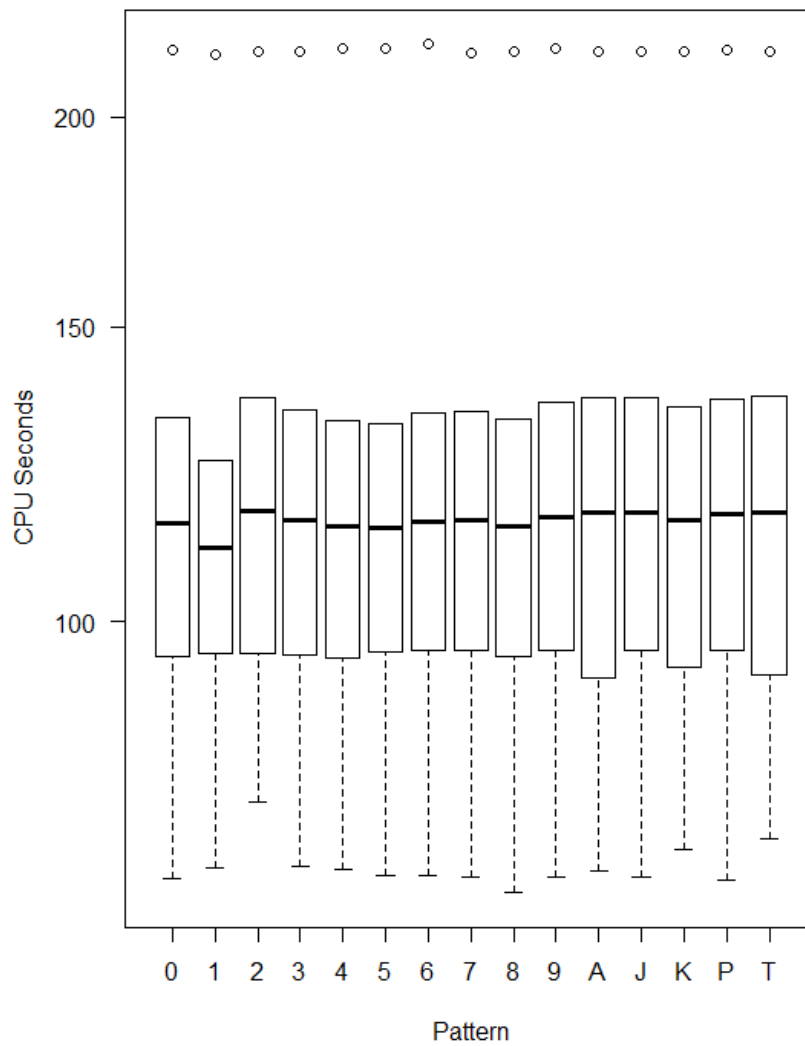


Fig. 4.8 The box plot presenting RetNet(8x5,1) simulation time for different input patterns at HPI infrastructure.

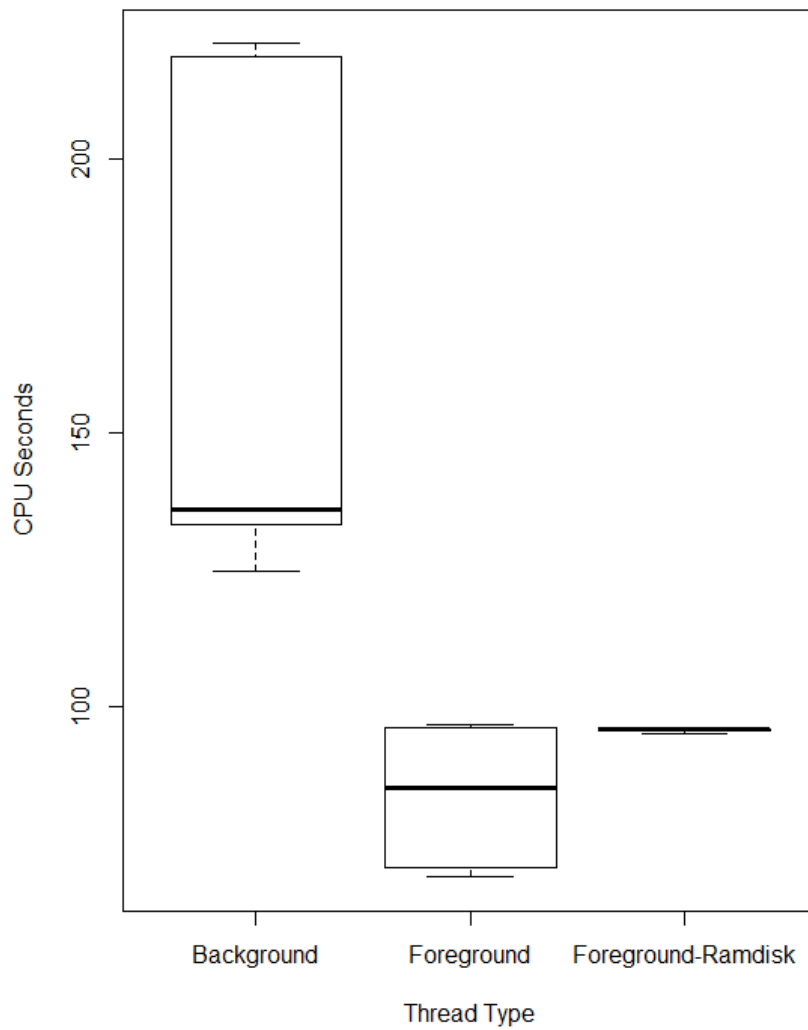


Fig. 4.9 The box plot presenting RetNet(8x5,1) simulation time for different thread types on HPI infrastructure.

(acting as a visual cortex, built of a single LSM column) was used in the next set of simulations at HPI and in AWS cloud to evaluate the NSP (presented in more details in Chapter 5).

In the simulated task, we used NSP to provide each model with three different stimulus patterns of “0”, “A”, “1” (through different values of NSP `$modelInput$` variable). This gave us the opportunity to evaluate the LSM system built with an increasing number of neurons in a standardised way. We simulated 1 second of this biological system (using the NSP variable `$simulationTime$`) across different execution environments.

The approach allowed to focus on the much larger models, with progressively larger liquid column. The structure of each column in the model is the same, but the size has been adjusted through the NSP variable `$columnDepth$`, and built in six versions:

1. RetNet(8x5,1,25) with 1040 neural cells placed in a rectangular cuboid of 8x5x25.
2. RetNet(8x5,1,50) with 2040 neural cells placed in a rectangular cuboid of 8x5x50.
3. RetNet(8x5,1,75) with 3040 neural cells placed in a rectangular cuboid of 8x5x75.
4. RetNet(8x5,1,100) with 4040 neural cells placed in a rectangular cuboid of 8x5x100.
5. RetNet(8x5,1,200) with 8040 neural cells placed in a rectangular cuboid of 8x5x200.
6. RetNet(8x5,1,300) with 12040 neural cells placed in a rectangular cuboid of 8x5x300.

4.5 Evaluating Liquid State Machine in RetNet(8x5,1)

4.5.1 Constructing LSM Readout Process

This subsection describes the data acquisition and pre-processing process that were executed to construct and evaluate readout for the LSM model. The data acquisition process is based on GENESIS [24] routines designed to write data from within the simulation environment to the operating system files. The `event_tofile` and `spikehistory` objects were used for recording spike event times to a file named through NSP variables (described in more detail in Section 5.4.2). The `SPIKESAVE` output routine was used. The details of that routine are described in GENESIS Manual¹. The simulation results are stored in .dat files, whereas any errors in .err, and outputs in .out files (e.g. RetNet40-0-retina.dat, RetNet40-0-column.dat, RetNet40.err, RetNet40.out).

Next, the data generated by the liquid column of the RetNet(8x5,1) model have been prepossessed in Data Science Studio (DSS) [114] and divided into the training set and test

¹GENESIS Manual <http://genesis-sim.org/GENESIS/Hyperdoc/Manual-7.html>.

Table 4.3 Neural Simulation Pipeline - Bare-bone Execution on HPI infrastructure.

On-Premise HPI					
Model	Neurons	Pattern	Spikes	Execution [s]	Memory [MB]
RetNet(8x5,1,25)	1040	0	20829	132.27	18.78
		A	20827	136.13	37.97
		1	20818	124.69	21.25
RetNet(8x5,1,50)	2040	0	61191	239.34	44.56
		A	61236	205.75	52.51
		1	61234	231.47	49.77
RetNet(8x5,1,75)	3040	0	121605	628.24	53.28
		A	121646	536.32	65.27
		1	121627	648.69	53.42
RetNet(8x5,1,100)	4040	0	202045	1269.55	88.81
		A	202036	1269.12	94.99
		1	202056	1321.30	72.24
RetNet(8x5,1,200)	8040	0	6512454	34082.88	1766.86
		A	6512440	34065.61	1791.56
		1	6568685	26548.51	1760.26
RetNet(8x5,1,300)	12040	0	14568366	102669.1	17482.09
		A	14568498	97643.23	17582.11
		1	14568473	101909.68	17169.67

Table 4.4 Neural Simulation Pipeline - Containerised Execution on HPI infrastructure.

Containerised HPI					
Model	Neurons	Pattern	Spikes	Execution [s]	Memory [MB]
RetNet(8x5,1,25)	1040	0	20834	45.66	26.37
		A	114427	185.05	46.51
		1	114403	214.94	71
RetNet(8x5,1,50)	2040	0	61215	150.40	57.97
		A	428368	1679.33	150.17
		1	428500	1659.42	146.53
RetNet(8x5,1,75)	3040	0	121659	495.38	63.01
		A	942366	3949.42	292.29
		1	942452	3957.19	80.70
RetNet(8x5,1,100)	4040	0	202014	938.60	73.91
		A	1656397	7253.15	139.56
		1	1656444	7467.51	437.87
RetNet(8x5,1,200)	8040	0	6512411	33000.14	1766.57
		A	6512452	33434.70	1766.35
		1	6512401	33198.74	437.87
RetNet(8x5,1,300)	12040	0	14568503	87163.24	4745.92
		A	14568712	86226.93	4810.72
		1	14457211	87213.56	4756.54

Table 4.5 Neural Simulation Pipeline - Containerised Execution on AWS Cloud Computing infrastructure.

Containerised AWS							
Model	Neurons	Pattern	Spikes	Execution [s]	Memory [MB]	Cost [USD]	
RetNet(8x5,1,25)	1040	0	114431	253.14	67.84	0.02	
		A	114425	253.53	46.07	0.02	
		1	114407	89.81	67.24	0.01	
RetNet(8x5,1,50)	2040	0	428380	427.58	128.9	0.10	
		A	428390	998.58	138.48	0.10	
		1	428454	980.13	130.53	0.09	
RetNet(8x5,1,75)	3040	0	942445	1879.22	280.51	0.22	
		A	942371	2033.78	291.59	0.19	
		1	942416	1737.16	273.31	0.17	
RetNet(8x5,1,100)	4040	0	1656346	3749.97	443.96	0.40	
		A	1656455	4104.38	436.08	0.39	
		1	1656388	3802.45	421.46	0.36	
RetNet(8x5,1,200)	8040	0	6512417	16913.23	1788.44	1.62	
		A	6512400	16564.92	1757.01	1.58	
		1	6512453	16850.31	1818.64	1.61	
RetNet(8x5,1,300)	12040	0	14568482	41814.32	17592.19	4.00	
		A	14568446	41916.18	17592.21	4.01	
		1	14568446	41824.63	17179.87	4.00	

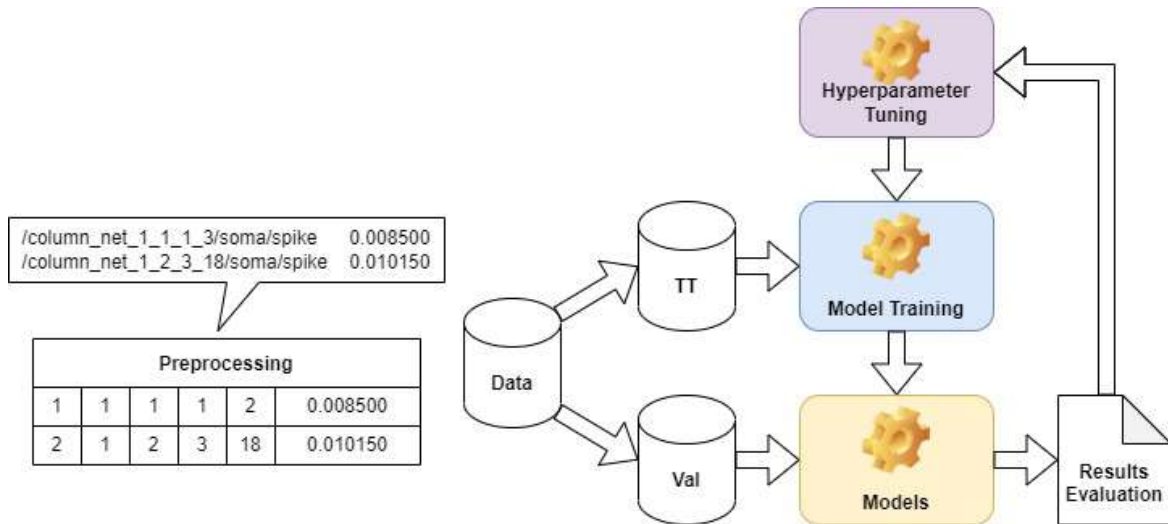


Fig. 4.10 Model selection during the LSM readout prototyping stage.

set using the 80/20 principle. 80% of randomly selected records have been selected into the training and testing set, whereas the remaining 20% become the validation set. The whole process of model selection during the LSM readout prototyping stage is presented in Fig. 4.10.

As mentioned in Subsection 2.8, the design of readout layer can be performed in three steps:

1. Identification of the neurons in the liquid column that should form the inputs to the readout layer.
2. Recording the time-varying states of the machine for different input patterns.
3. Applying a supervised learning algorithm for the pairs in order to train the readout layer in such a way that its response maximises similarity to the desired output.

In case of this experimental stage of the thesis, all the neurons in the column were used, and five numeric input features were derived from the dataset generated through simulation of 1 second as the biological system, that was repeated 3 times. The sixth feature, that is an input pattern (e.g. 0, A, 1), was defined for the simulation as an input parameter through *NSP variables* described in Section 5.4.2. As a result the LSM generated a total of 11.8 MB of spiking data points, that are organised in 273519 distinct spike observations, as per Fig. 4.10.

The numeric avg-std re-scaling has been performed prior to training using scikit-learn [1], as there were large differences in the absolute values of the features in the dataset. Standard re-scaling scales the feature to a standard deviation of 1 and a mean of 0 to improve model

performance. This approach was not selected for algorithms that are based on decision trees, as re-scaling has no effect on decision trees.

4.5.2 Machine Learning Algorithms

Author has evaluated twelve supervised learning algorithms to construct the optimal readout process. The implementation of these algorithms comes from Scikit Learn [1], Xgboost [36] and LightGBM [126]. There are: Light Gradient-boosting Machine (LightGBM), Gradient Boosted Trees, eXtreme Gradient Boosting (XGBoost), Extra Trees, Random Forest, Logistic Regression, Stochastic Gradient Descent (SGD), Multi-layer Perceptron (MLP), LASSO-LARS, Support Vector Machines (SVM), Decision Tree, and AdaBoost.

Table 4.6 provides a total training time for each supervised learning algorithm. All models have been trained using DSS². The total training time includes the time needed for the following steps: loading train set, loading test set, collecting statistics, pre-processing train set, pre-processing test set, fitting the model, saving the model, scoring the modes. The vast majority of time was spent on fitting each model; all the remaining tasks always took approximately a second or less. The following subsection provides an overview of all the experiment evaluation metrics.

4.5.3 Experiment Metrics

The problem that the LSM readout solves is classification. Although, the dataset described in 4.5.1 is balanced, the classification will be evaluated through a few key, computed as follows:

- Accuracy metric:

$$A = \frac{TP + TN}{N}, \quad (4.1)$$

- Precision metric:

$$P = \frac{TP}{TP + FP}, \quad (4.2)$$

- Recall (Sensitivity) metric:

$$R = \frac{TP}{TP + FN}, \quad (4.3)$$

- F1 Score:

$$F_1 = 2 \frac{\pi \times \rho}{\pi + \rho} \quad (4.4)$$

²DSS at PJAIT: <http://dataiku.pjwstk.edu.pl:11000/>.

where

$$\pi = \frac{TP}{TP + FP}, \quad (4.5)$$

$$\rho = \frac{TP}{TP + FN}. \quad (4.6)$$

N is the number of observations, TP is the number of true positives, TN is the number of true negatives, FP is the number of false positives and FN the number of false negatives. As the algorithms selected for evaluation produce prediction probabilities, the Multi-class Area under the ROC curve (AUC or MAUC) scores is also presented.

The first classification metric is Accuracy, that is a ratio of correctly predicted observations to the total observations. Precision is the ratio of correctly predicted positive observations to the total predicted positive observations. Recall indicates the ratio of correctly predicted positive observations to the all observations in the actual class.

Finally, in author's view the most important F1 Score, that is an error metric calculating model performance using the harmonic mean of Precision and Recall for the minority positive class. Importance of F1 Score is derived from fact that it returns accurate results for both balanced and imbalanced datasets. F1 score can be interpreted as a single measure of overall model performance ranging from 0 to 1, where 1 is the best result. In other words, F1 Score indicates model's balanced classification ability to capture both the positive cases (Recall), while being accurate with the cases it does not capture (Precision). For the purpose of this thesis the interpretation of F1 Score is as follows (1) > 0.9 Excellent (2) 0.8 - 0.9 Very good (3) 0.5 - 0.8 Good, (4) < 0.5 Not good.

All the evaluation metrics that are reported in Table 4.6 were calculated using the 10-fold Cross-Validation method [75] using scikit-learn [1, 180]. This method was selected to ensure a proper quality of results for all the twelve classifiers reported in this chapter. As one can see in Table 4.6 our LSM readout using *LightGBM* [126] algorithm consistently achieves the best classification accuracy of 81% (the exact accuracy of 0.8096). For the current LSM design, there are also two other methods that consistently achieve the results over 66% for the three-class classification problem. These are *Gradient Boosted Trees* [73] and *XGBoost* [36] methods (the exact accuracy of 0.6896 and 0.6653 respectively). The results in Table 4.6 have been ordered descending by the F1 Score, to highlight the overall best results in the top rows of the table.

The subsequent subsections, from the Subsection 4.5.4 to 4.5.15 provide a deep-dive into performance of each of the twelve algorithms evaluated, that is combined with an overview of the parameters used to achieve these results, as well as a few additional figures. A single figure that is presented for every algorithm is the Receiver Operating Characteristic (ROC) curve [85]. It was done to allow for an easy (visual) comparison all the classification results.

Table 4.6 Summary of LSM Readout classification performance for RetNet(5x8,1) seeing '0','1','A' after 10-fold cross-validation.

Algorithm	Accuracy	Precision	Recall	F1 Score	ROC AUC	Training [s]
LightGBM	0.8096	0.8099	0.8096	0.8095	0.9483	1082.6
Gradient Boosted Trees	0.6896	0.6911	0.6896	0.6900	0.8695	359.7
XGBoost	0.6653	0.6664	0.6658	0.6653	0.8533	655.2
Extra Trees	0.5061	0.5118	0.5066	0.5057	0.6946	124.1
Random Forest	0.4754	0.4787	0.4756	0.4751	0.6796	221.0
Logistic Regression	0.3563	0.3563	0.3566	0.3563	0.5293	52.3
SGD	0.3570	0.3574	0.3573	0.3551	0.5290	45.4
Multi-layer Perceptron	0.3618	0.3625	0.3610	0.3548	0.5364	125.7
LASSO-LARS	0.3551	0.3552	0.3543	0.3528	0.5294	531.8
SVM	0.3564	0.3591	0.3580	0.3489	0.5330	160978.8
Decision Tree	0.3819	0.4649	0.3813	0.3283	0.5590	21.7
AdaBoost	0.3708	0.3715	0.3703	0.3692	0.5456	118.6

The curve shows the true positive rate (Sensitivity) on the Y axis, against the false positive rate (Specificity) on the X axis. The interpretation of the figure is that the "faster" the curve climbs, the better a classifier is. On the other hand, the closer to the diagonal line the curve is, the worse a classifier is. Such an interpretation of "steepness" of the ROC curves is coined, because for a given classifier it is optimal to maximise the true positive rate, while minimising the false positive rate. The Figures from 4.11b to 4.22b exhibit each ROC curve for each of the LSM readout supervision methods presented in Table 4.6. As these ROC curves were drawn for a multi-label classification task, it was necessary to binarize the output, so that a single ROC curve could be drawn by considering each element of the label indicator matrix through micro-averaging, which gives equal weight to the classification of each label. For the purpose of this thesis the interpretation of Multi-class Area Under the Curve is as follows (1) > 0.9 Excellent (2) 0.8 - 0.9 Very good (3) 0.6 - 0.8 Good, (4) < 0.6 Not good.

To summarise, author decided to calculate and report the following classification metrics [143]: Accuracy, Precision, Recall, ROC - AUC Score, F1 Score; that were complemented with the Log loss, Calibration loss and Hamming loss. All of them are reported in the subsequent subsections from 4.5.4 to 4.5.15.

Table 4.7 Model parameters and detailed performance metrics for LSM readout using LightGBM algorithm.

Parameter	Value	Metric	Value
Booster	gbdt	Precision	0.8099 (\pm 0.0070)
Actual number of trees	76	Accuracy	0.8096 (\pm 0.0071)
Maximum number of leaves	31	Recall	0.8096 (\pm 0.0070)
Learning rate	0.2	ROC - MAUC Score	0.9483 (\pm 0.0023)
α (L1 regularisation)	0	F1 Score	0.8095 (\pm 0.0070)
λ (L2 regularisation)	0	Log loss	0.4564 (\pm 0.0083)
Minimal gain to perform a split on a leaf	0	Hamming loss	0.1904 (\pm 0.0071)
Min sum of instance weight in a child	0.001		
Subsample ratio of the training instance	1		
Fraction of columns in each tree	0.9		

Table 4.8 Confusion matrices for LSM readout using LightGBM algorithm (% of predicted classes vs. % of actual class).

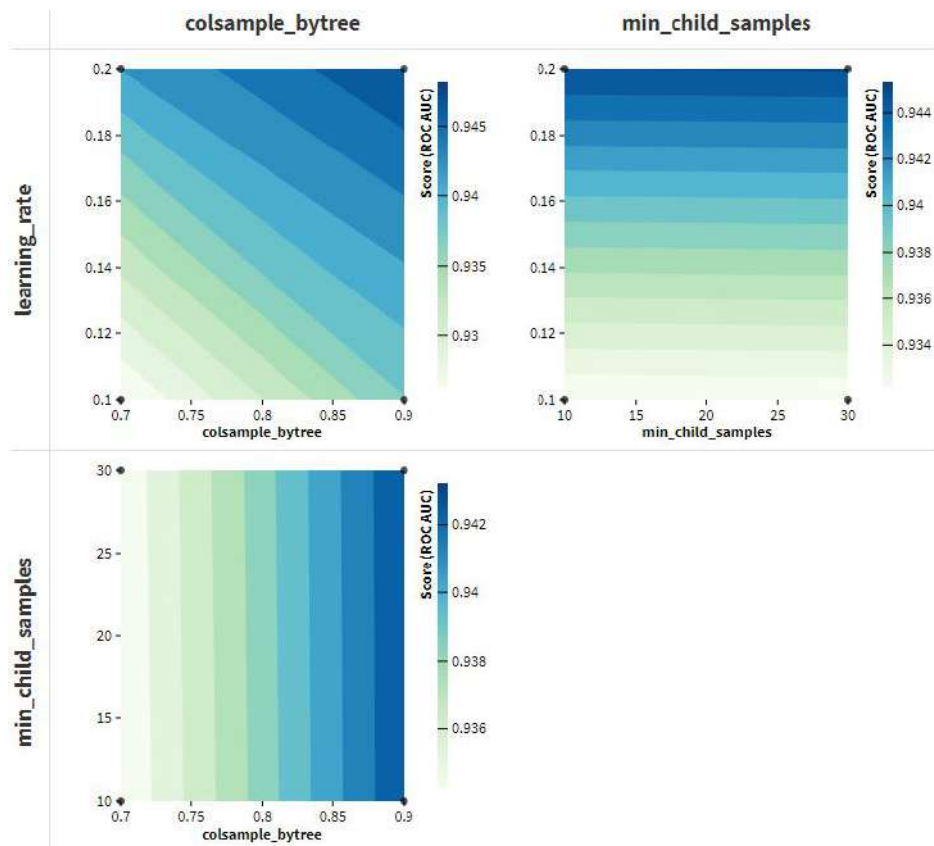
<i>Act. \ Pred.</i>	0	A	1	<i>Act. \ Pred.</i>	0	A	1	Total
0	80%	7%	10%	0	83%	7%	10%	100%
A	9%	85%	12%	A	10%	79%	12%	100%
1	10%	8%	79%	1	11%	8%	81%	100%
Total	100%	100%	100%					

4.5.4 Detailed Results for LightGBM

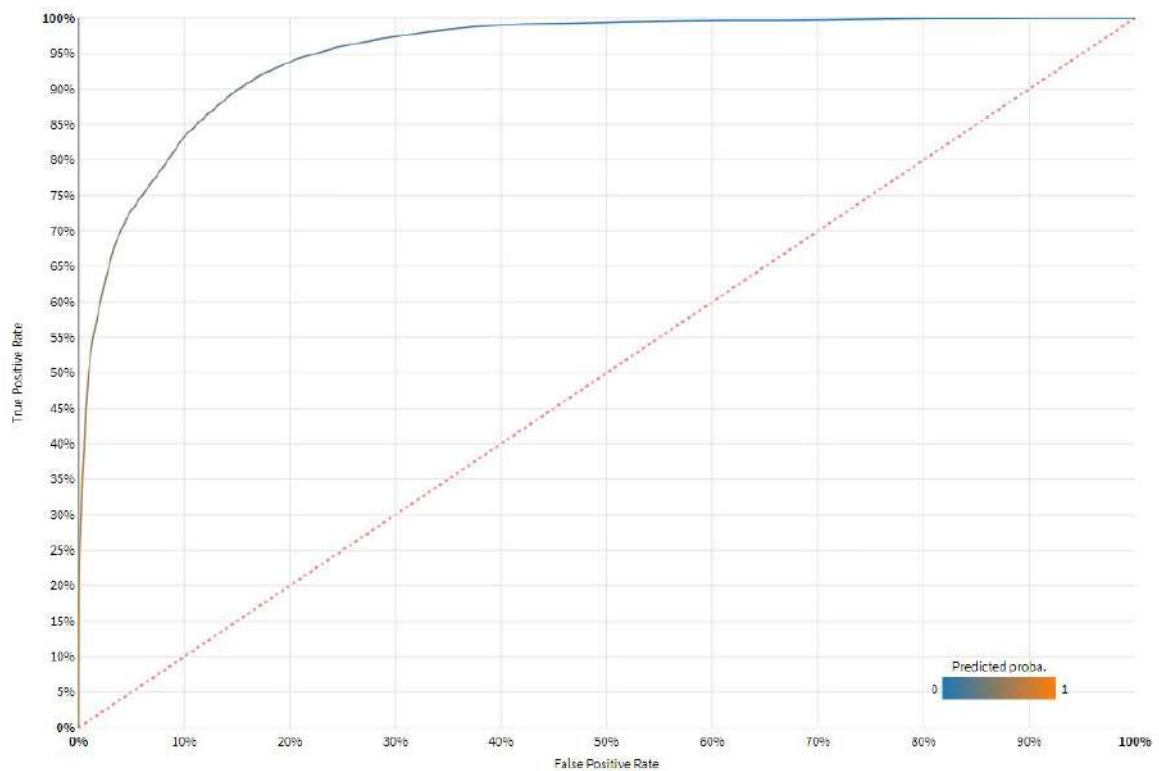
The LightGBM readout was calculated using a tree-based gradient boosting library [126] designed by Microsoft³ to be “distributed and efficient”. It provides three distributed learning algorithms: data parallel, feature parallel, and voting parallel, and uses the leaf-wise tree growth algorithm.

The calculated Accuracy is 0.8096 (81%), while the F1 Score is 0.8095 (\pm 0.0070), that is a *very good result*. The calculated MAUC for the LightGBM LSM readout is 0.948 (\pm 0.002), which is an *excellent result*.

³LightGBM Documentation <https://lightgbm.readthedocs.io/en/v3.3.2/>.



(a) LightGBM hyperparameter search pairwise dependency plots.



(b) LightGBM Receiver Operating Characteristic curve. The AUC for class '0' is 0.950.

Fig. 4.11 Additional performance and model details for LSM readout using LightGBM algorithm.

Table 4.9 Model parameters and detailed performance metrics for LSM readout using Gradient Boosted Trees algorithm.

Parameter	Value	Metric	Value
Loss	Deviance	Precision	0.6911 (\pm 0.0152)
Feature sampling strategy	Default (friedman_mse)	Accuracy	0.6896 (\pm 0.0143)
Number of boosting stages	100	Recall	0.6896 (\pm 0.0141)
Eta (learning rate)	0.1	ROC - MAUC Score	0.8695 (\pm 0.0046)
Max trees depth	3	F1 Score	0.6897 (\pm 0.0145)
Minimum samples at leaf	1	Log loss	0.8334 (\pm 0.0045)
		Hamming loss	0.3104 (\pm 0.0143)

Table 4.10 Confusion matrices for LSM readout using Gradient Boosted Trees algorithm (% of predicted classes vs. % of actual class).

<i>Act. \ Pred.</i>	0	A	1	<i>Act. \ Pred.</i>	0	A	1	Total
0	67%	15%	14%	0	71%	16%	13%	100%
A	18%	69%	14%	A	19%	68%	13%	100%
1	15%	16%	71%	1	16%	17%	67%	100%
Total	100%	100%	100%					

4.5.5 Detailed Results for Gradient Boosted Trees

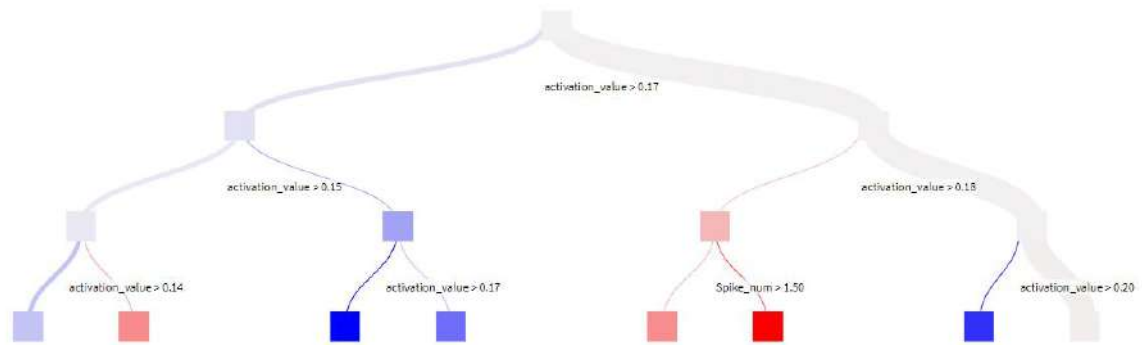
The Gradient boosted trees (GBT) readout was calculated using the GBT ensemble method based on decision trees implemented in scikit-learn [180]. In this approach the trees are added to the ensemble in a sequential way, with each tree attempting to improve the overall classification performance of the model⁴.

The calculated Accuracy for the GBT readout is 0.6896 (69%), while the F1 Score is 0.6897 (\pm 0.0145), that is a *good result*. The calculated MAUC for the GBT LSM readout is 0.870 (\pm 0.005), which is a *very good result*.

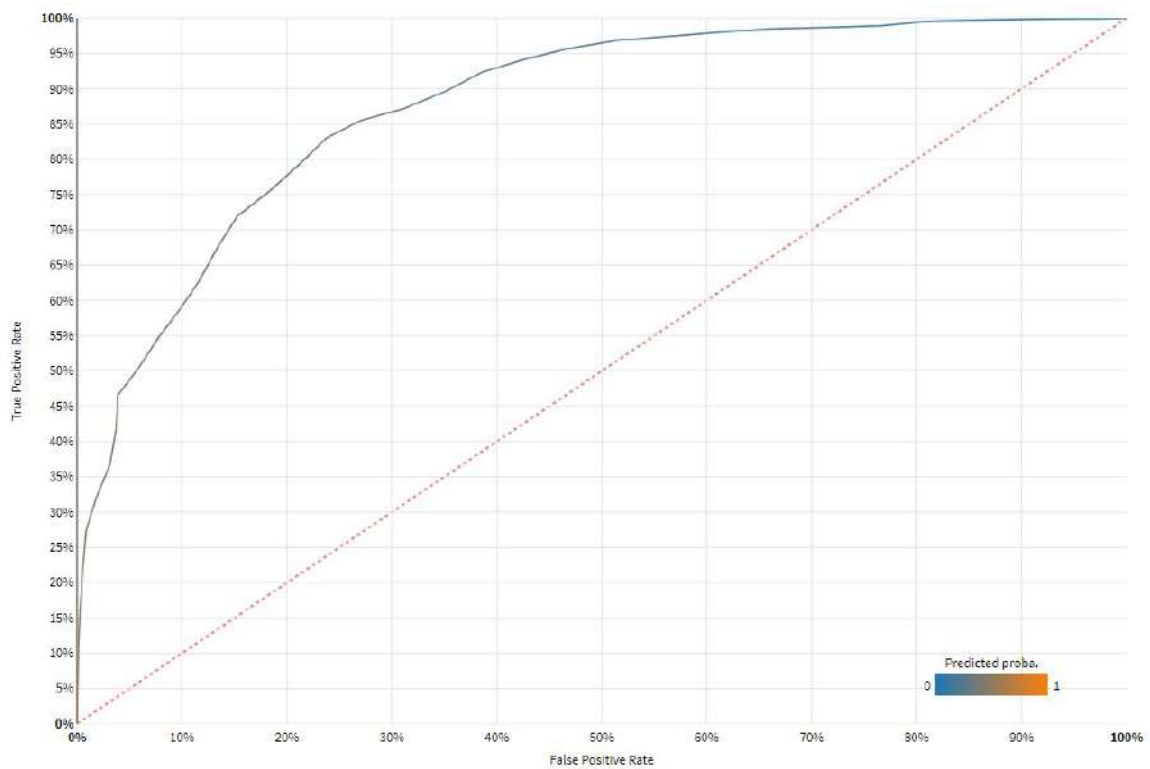
4.5.6 Detailed Results for XGBoost

XGBoost is an advanced gradient boosted tree algorithm [36]. It supports parallel computations, regularisation, early stopping which makes it a fast, scalable and accurate algorithm, that is also implemented as a package for several programming languages.

⁴Gradient Boosted Trees Documentation and Parameters <https://scikit-learn.org/stable/modules/ensemble.html>.



(a) Sample GBT giving prediction of 1.81 for $spike_num > 1.50$. Samples count 1599 (0.73%), class '0'.



(b) Gradient Boosted Trees Receiver Operating Characteristic curve. The AUC for class '0' is 0.876.

Fig. 4.12 Additional details and model performance for LSM readout using Gradient Boosted Trees algorithm.

Table 4.11 Model parameters and detailed performance metrics for LSM readout using XGBoost algorithm.

Parameter	Value	Metric	Value
Booster	gbtree	Precision	0.6664 (± 0.0126)
Actual number of trees	53	Accuracy	0.6653 (± 0.0138)
Max trees depth	3	Recall	0.6658 (± 0.0134)
Eta (learning rate)	0.2	ROC - MAUC Score	0.8533 (± 0.0140)
α (L1 regularisation)	0	F1 Score	0.6653 (± 0.0140)
λ (L2 regularisation)	1	Log loss	0.8582 (± 0.0356)
γ (Min loss reduction to split a leaf)	0	Hamming loss	0.3347 (± 0.0138)
Min sum of instance weight in a child	1		
Subsample ratio of the training instance	1		
Fraction of columns in each tree	1		
Replace missing values with	NaN		

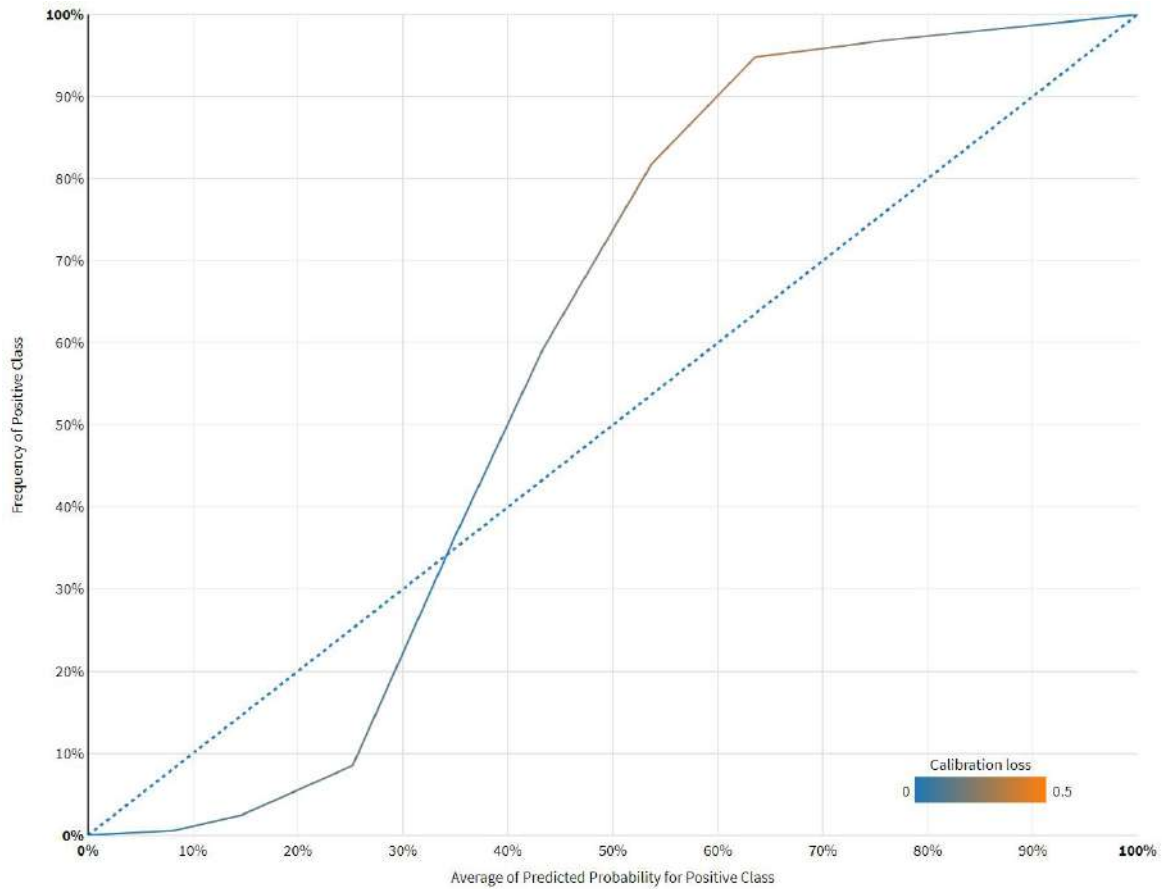
The calculated Accuracy for the XGBoost readout is 0.6653 (67%), while the F1 Score is 0.6653 (± 0.0140), that is a *good result*. The calculated MAUC for the XGBoost LSM readout is 0.8533 (± 0.0140), which is a *very good result*.

Figure 4.13a presents the calibration loss of predicted probabilities for the class '0', so the first of three classes evaluated by the algorithm. In this context the calibration indicates the consistency between predicted probabilities and their real frequencies in the test dataset, so that a perfectly calibrated readout will have a calibration curve equal to the diagonal line. However, in real life scenarios the calibration curve is always distinct from the diagonal line [19]. As the average distance between the curve and the diagonal line, averaged over the test set, and weighted by the number of elements is treated as a measure of the quality of the calibration our calibration line indicates the average calibration of the XGBoost algorithm⁵.

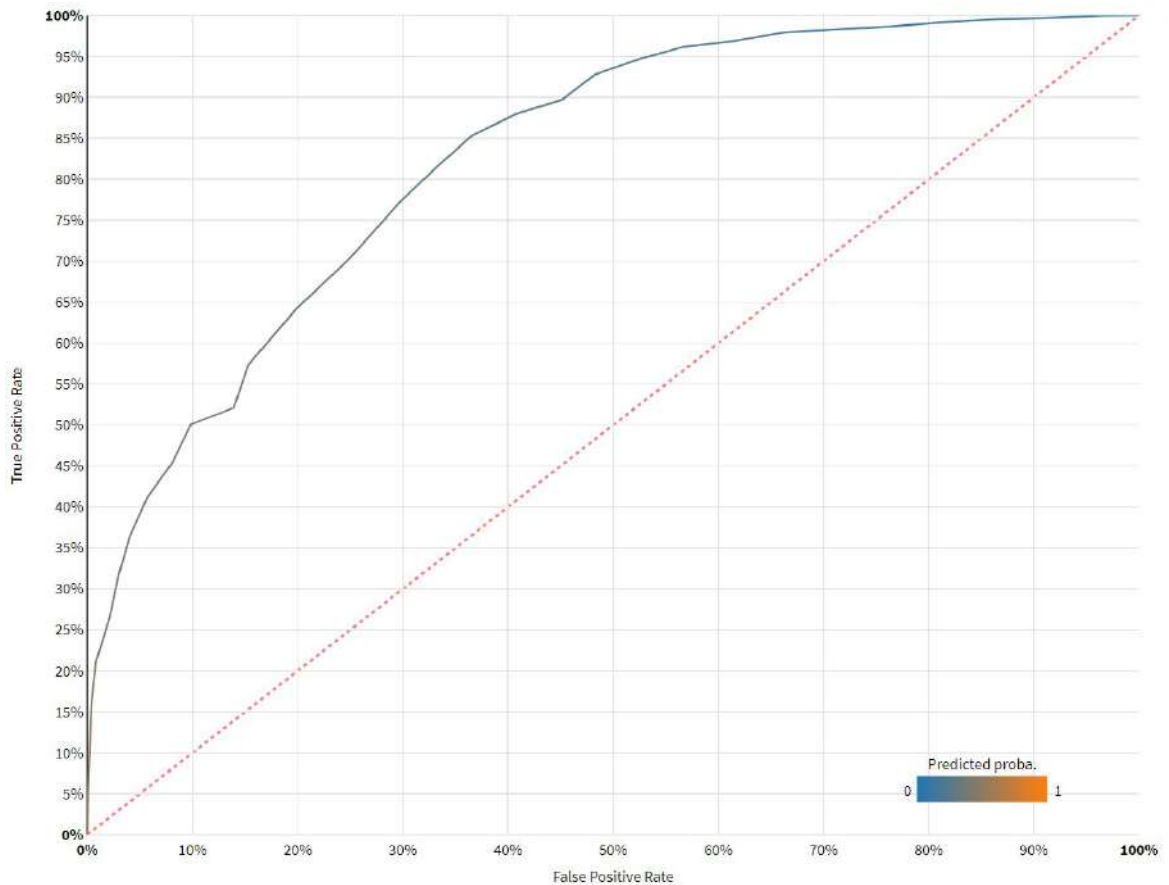
4.5.7 Detailed Results for Extra Trees

Extra Trees, similarly to Gradient Boosted Trees and Random Forests, are an ensemble methods of Machine Learning. In contrast to other ensemble algorithms, apart from feature sampling at each stage of splitting the tree, the method also samples random threshold at

⁵XGBoost Documentation and Parameters <https://xgboost.readthedocs.io/en/stable/index.html>.



(a) XGBoost calibration curve. The calibration loss for class '0' is 0.1037.



(b) XGBoost Receiver Operating Characteristic curve. The AUC for class '0' is 0.827.

Fig. 4.13 Additional model performance details for LSM readout using XGBoost algorithm.

Table 4.12 Confusion matrices for LSM readout using XGBoost algorithm (% of predicted classes vs. % of actual class).

<i>Act. \ Pred.</i>	0	A	1	<i>Act. \ Pred.</i>	0	A	1	Total
0	60%	17%	19%	0	66%	15%	19%	100%
A	24%	65%	18%	A	25%	57%	18%	100%
1	16%	18%	64%	1	17%	17%	66%	100%
Total	100%	100%	100%					

Table 4.13 Model parameters and detailed performance metrics for LSM readout using Extra Trees algorithm.

Parameter	Value	Metric	Value
Number of trees	100	Precision	0.5118 (± 0.0210)
Max trees depth	8	Accuracy	0.5061 (± 0.0200)
Min samples per leaf	1	Recall	0.5066 (± 0.0199)
Min samples to split	3	ROC - MAUC Score	0.6946 (± 0.0150)
Split quality criterion	Gini	F1 Score	0.5057 (± 0.0204)
Use bootstrap	Yes	Log loss	1.0607 (± 0.0034)
Feature sampling strategy	auto	Hamming loss	0.4939 (± 0.0200)

which to make the splits. This is because the additional randomness might improve the generalisation capabilities of the model⁶.

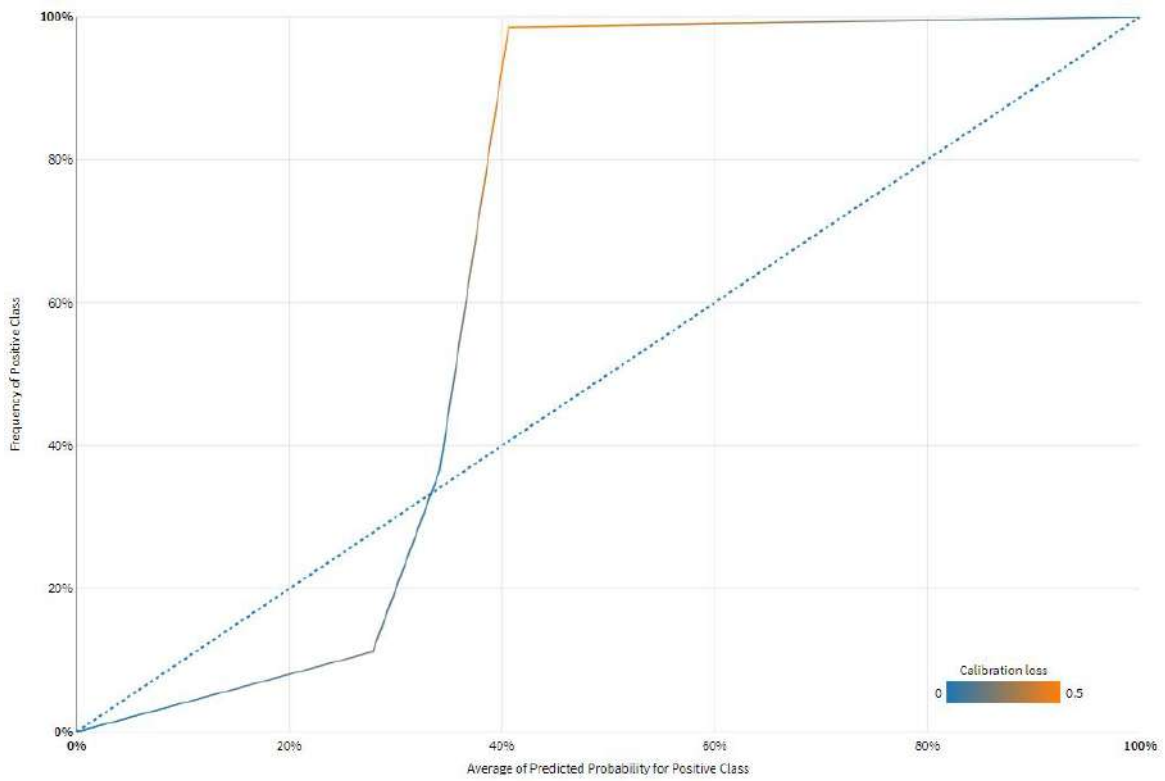
The calculated Accuracy for the Extra Trees readout is 0.5061 (51%), while the F1 Score is 0.5057 (± 0.0204), that is still a *good result*. The calculated MAUC for the Extra Trees based LSM readout is 0.695 (± 0.015), which is a good result.

4.5.8 Detailed Results for Random Forest

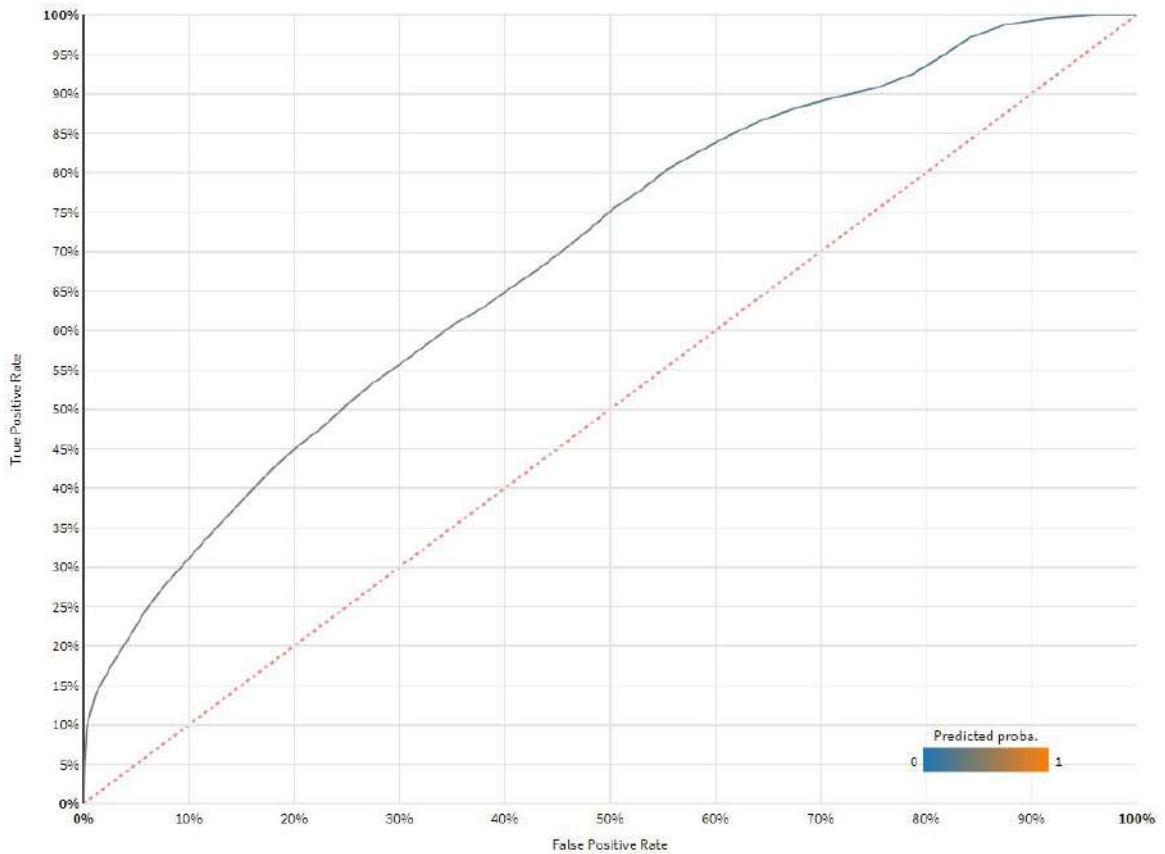
Random Forest decision tree is a simple classification algorithm which creates a decision tree in the way that each node of the decision tree includes a condition on one of the input features⁷. Random Forests are considered to deliver good classification results, at the cost of explainability of the model. As a result, figures like Fig. 4.15a presenting a sample tree

⁶Extra Trees Documentation and Parameters <https://scikit-learn.org/stable/modules/ensemble.html>.

⁷Random Forest Documentation and Parameters <https://scikit-learn.org/stable/modules/ensemble.html>.



(a) Extra Trees calibration curve. The calibration loss for class '0' is 0.0449.



(b) Extra Trees Receiver Operating Characteristic curve. The AUC for class '0' is 0.695.

Fig. 4.14 Additional model performance details for LSM readout using Extra Trees algorithm.

Table 4.14 Confusion matrices for LSM readout using Extra Trees algorithm (% of predicted classes vs. % of actual class).

<i>Act. \ Pred.</i>	0	A	1	<i>Act. \ Pred.</i>	0	A	1	Total
0	47%	20%	27%	0	58%	17%	25%	100%
A	26%	58%	23%	A	32%	48%	21%	100%
1	27%	22%	50%	1	34%	19%	47%	100%
Total	100%	100%	100%					

Table 4.15 Model parameters and detailed performance metrics for LSM readout using Random Forest algorithm.

Parameter	Value	Metric	Value
Number of trees	100	Precision	0.4787 (\pm 0.0214)
Max trees depth	12	Accuracy	0.4754 (\pm 0.0205)
Min samples per leaf	10	Recall	0.4756 (\pm 0.0206)
Min samples to split	30	ROC - MAUC Score	0.6796 (\pm 0.0177)
Split quality criterion	Gini	F1 Score	0.4751 (\pm 0.0204)
Use bootstrap	Yes	Log loss	1.0289 (\pm 0.0057)
Feature sampling strategy	auto	Hamming loss	0.5246 (\pm 0.0205)

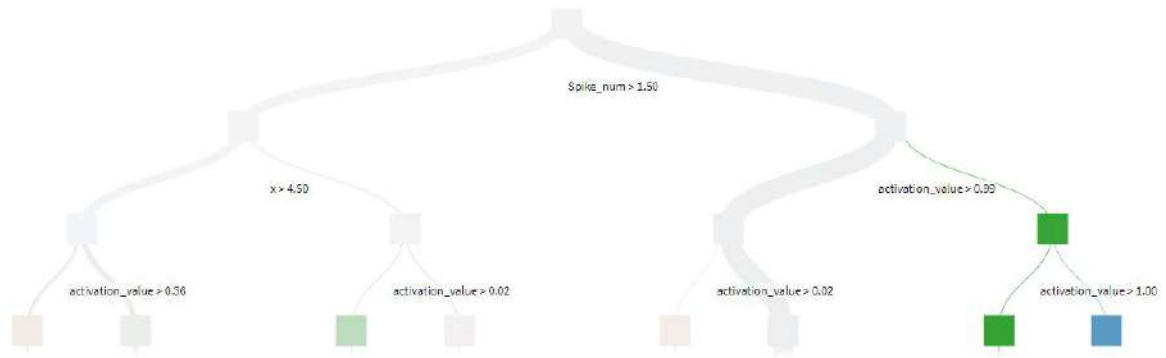
giving a prediction of '1' with 97.36% certainty, are often created. All the results for this algorithm are based on the scikit-learn [180] implementation of the algorithm.

The Accuracy calculated for the Random Forest readout is 0.4754 (48%), while the F1 Score is 0.4751 (\pm 0.0204), that is *not a good result*. The MAUC for the Random Forest LSM readout is 0.685 (\pm 0.018), which is still a good result.

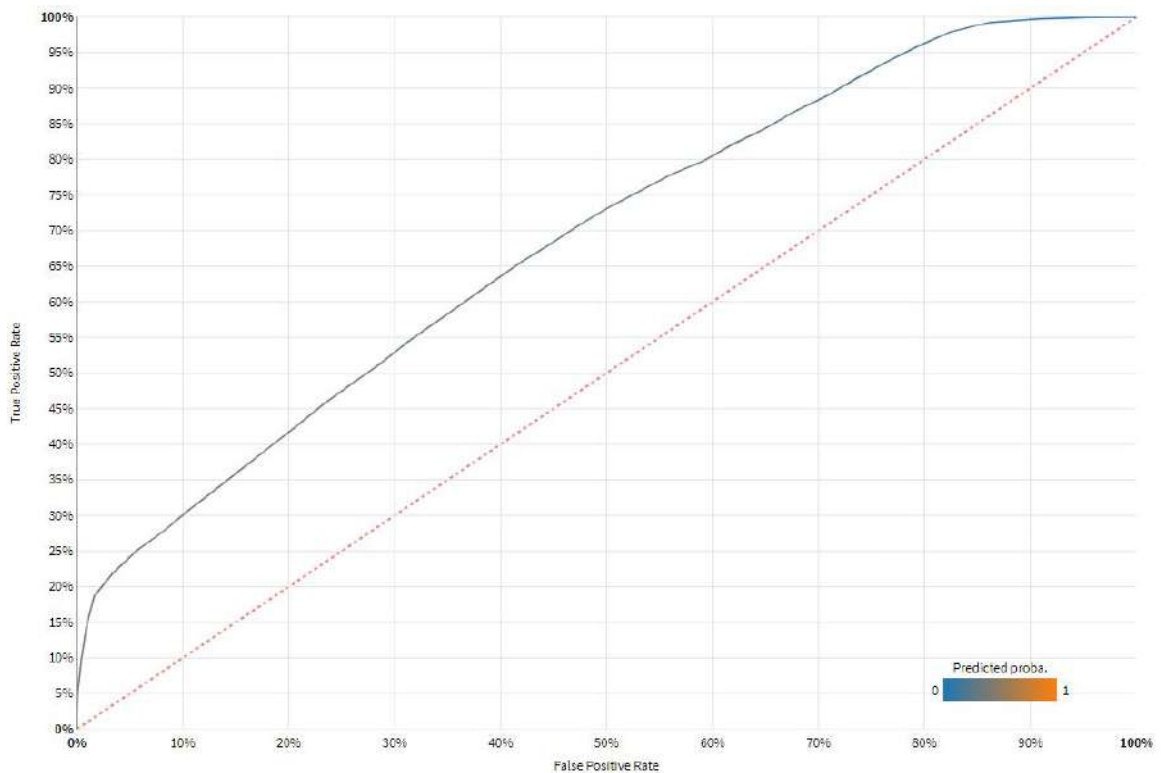
4.5.9 Detailed Results for Logistic Regression Max Entropy

The Logistic Regression or Max Entropy readout algorithm was computed using the scikit-learn [180] linear model. The algorithm calculates the target feature as a linear combination of the input features⁸. The algorithm minimises either a logit or sigmoid function, to allow for classification. As the algorithm is susceptible to over-fitting and sensitive input errors, the regularisation on weights is often used [240].

⁸Logistic Regression Documentation and Parameters https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html.



(a) Sample Random Forest tree giving prediction of '1' (97.36%), '0' (2.64%), and A (0%) based on decision rule $spike_num > 1.50$ and $activation_value \in (0.99, 1.00>$. Samples count is 615 (0.45%).



(b) Random Forest Receiver Operating Characteristic curve. The AUC for class '0' is 0.685.

Fig. 4.15 Additional model details and performance for LSM readout using Random Forest algorithm.

Table 4.16 Confusion matrices for LSM readout using Random Forest algorithm (% of predicted classes vs. % of actual class).

<i>Act. \ Pred.</i>	0	A	1	<i>Act. \ Pred.</i>	0	A	1	Total
0	45%	23%	27%	0	55%	20%	25%	100%
A	28%	53%	24%	A	33%	44%	22%	100%
1	27%	24%	48%	1	34%	21%	46%	100%
Total	100%	100%	100%					

Table 4.17 Model parameters and detailed performance metrics for LSM readout using Logistic Regression algorithm.

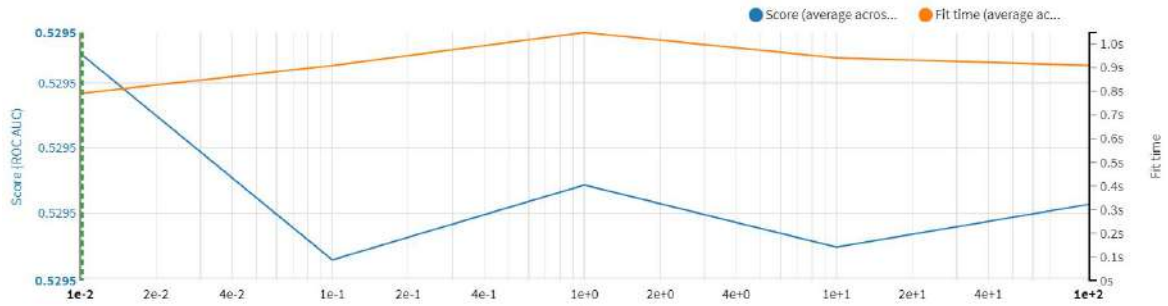
Parameter	Value	Metric	Value
Penalty	L2	Precision	0.3563 (\pm 0.0045)
C	0.01	Accuracy	0.3563 (\pm 0.0046)
		Recall	0.3566 (\pm 0.0045)
		ROC - MAUC Score	0.5293 (\pm 0.0042)
		F1 Score	0.3563 (\pm 0.0046)
		Log loss	1.0962 (\pm 0.0008)
		Hamming loss	0.6437 (\pm 0.0046)

The Accuracy calculated for the Logistic Regression readout is 0.3563 (36%), while the F1 Score is 0.3563 (\pm 0.0046), that is *not a good result*. The MAUC for the Logistic Regression LSM readout is 0.529 (\pm 0.004), which is *not a good result*.

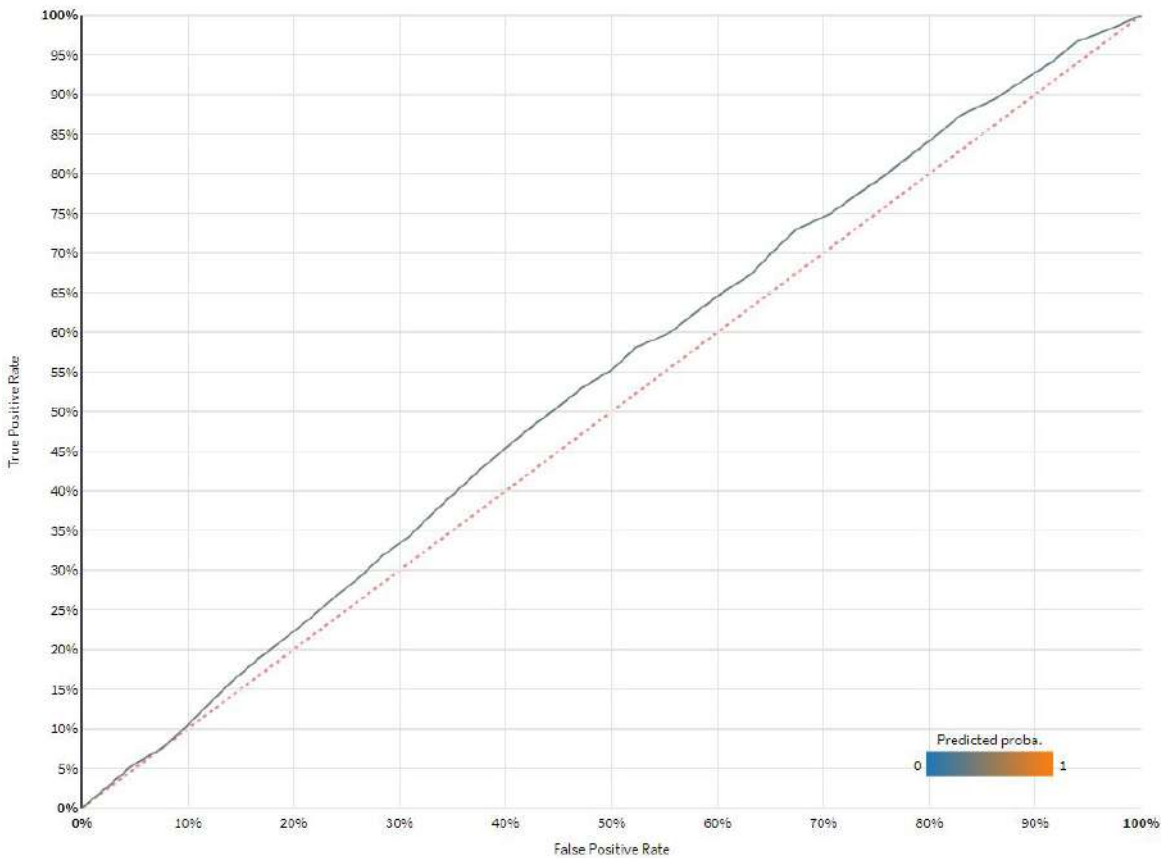
4.5.10 Detailed Results for Logistic Regression Stochastic Gradient Descent

The Stochastic Gradient Descent (SGD) algorithm [20] is a simple and often efficient way to fit linear classifiers under convex loss functions such as linear Support Vector Machines or Logistic Regression. SGD is an optimisation technique, rather than a specific family of machine learning models, that uses several parameters⁹. In this particular case our LSM readout uses logistic regression, which is fitted using SGD instead of being trained by a cross-entropy loss function and newton-cg solver; the case that is detailed in Subsection 4.5.9. The results for SGD were computed using the scikit-learn [180] package. A basic hyper-

⁹SGD trained Logit Documentation and Parameters <https://scikit-learn.org/stable/modules/sgd.html>.



(a) Logistic Regression hyperparameter optimization. The model was trained using a grid search on 5 combinations of parameter C: 0.01, 0.1, 1, 10, 100. The Plot presents averages across other dimensions.



(b) Logistic Regression Receiver Operating Characteristic curve. The AUC for class '0' is 0.534.

Fig. 4.16 Additional model details and performance for LSM readout using Logistic Regression algorithm.

Table 4.18 Confusion matrices for LSM readout using Logistic Regression algorithm (% of predicted classes vs. % of actual class).

<i>Act.</i> \ <i>Pred.</i>	0	A	1	<i>Act.</i> \ <i>Pred.</i>	0	A	1	Total
0	36%	33%	31%	0	36%	32%	32%	100%
A	34%	35%	32%	A	34%	33%	33%	100%
1	30%	32%	36%	1	31%	31%	37%	100%
Total	100%	100%	100%					

Table 4.19 Model parameters and detailed performance metrics for LSM readout using Stochastic Gradient Descent.

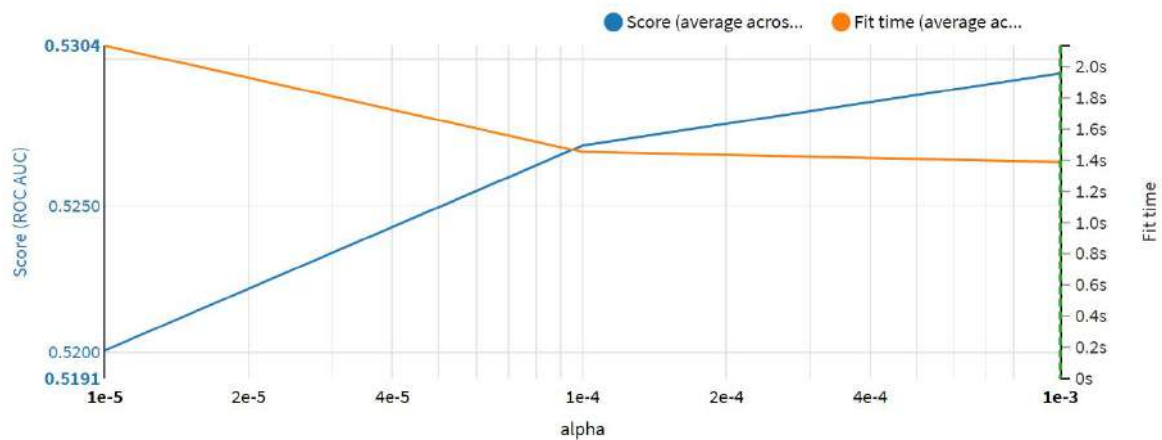
Parameter	Value	Metric	Value
Loss function	log	Precision	0.3574 (\pm 0.0093)
Penalty	l1	Accuracy	0.3570 (\pm 0.0084)
Stopping tolerance	0.001	Recall	0.3573 (\pm 0.0093)
Max iterations	1000	ROC - MAUC Score	0.5290 (\pm 0.0052)
Actual iterations	7	F1 Score	0.3551 (\pm 0.0078)
		Log loss	1.0964 (\pm 0.0009)
		Hamming loss	0.3551 (\pm 0.0078)

parameter optimisation was applied using a grid search on five combinations of α parameter, as present in Fig. 4.17a.

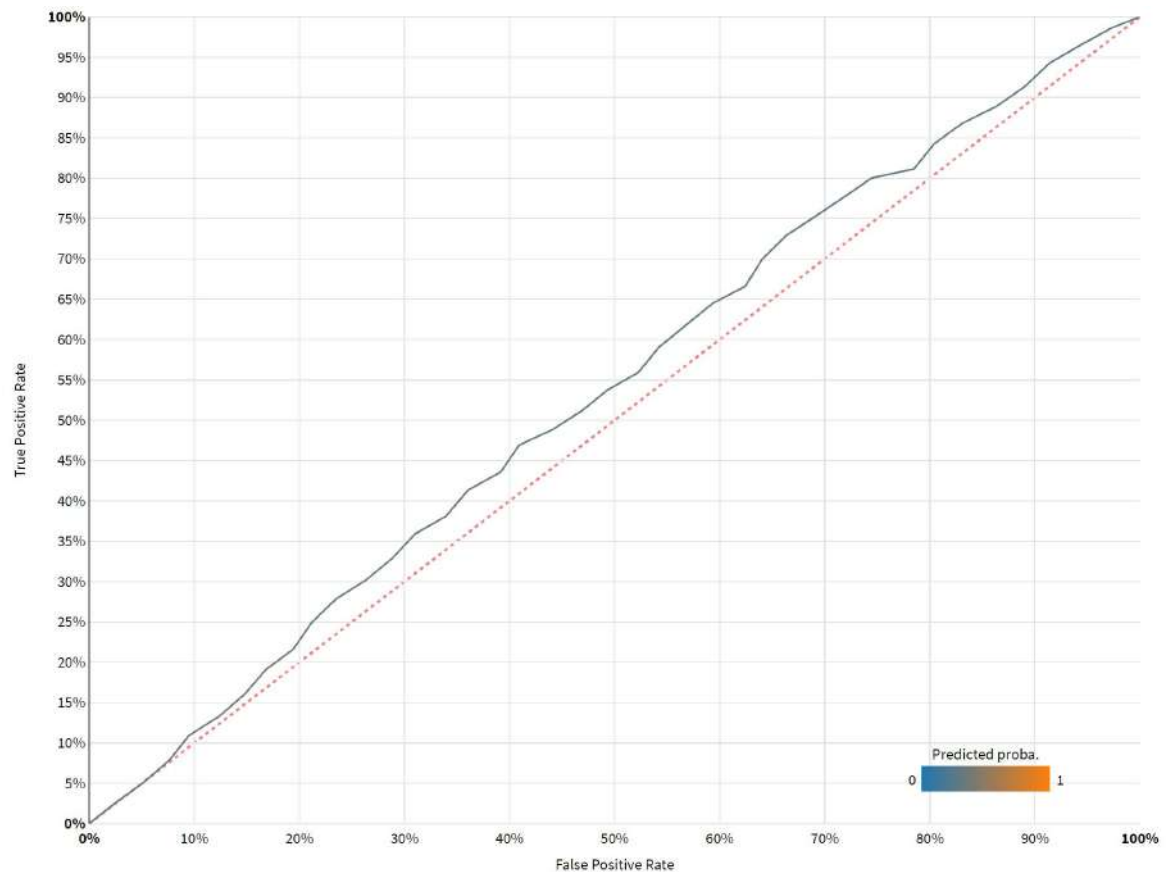
The Accuracy calculated for the SGD trained logit readout is 0.3570 (36%), while the F1 Score is 0.3551 (\pm 0.0078), that is *not a good result*. The MAUC the Stochastic Gradient Descent train logit LSM readout is 0.529 (\pm 0.005), which is *not a good result*.

4.5.11 Detailed Results for Multi-layer Perceptron

Artificial neural networks like Multi-layer Perceptron (MLP) are the class of ML models which are inspired by the functioning of neurons. These networks are built with several so called *hidden layers* of neurons, which receive inputs and transmit them to the next layer. MLP mixes the inputs, allowing for complex decision scenarios, as well as using several interesting and well documented gradient-descent procedures to adjust the weights [105].



(a) Stochastic Gradient Descent hyperparameter optimization. This model was trained using a grid search on 3 combinations of parameter alpha: 0.00001, 0.0001, 0.001. The Plot presents averages across other dimensions.



(b) Stochastic Gradient Descent Receiver Operating Characteristic curve. The AUC for class '0' is 0.535.

Fig. 4.17 Additional model details and performance for LSM readout using Stochastic Gradient Descent algorithm.

Table 4.20 Confusion matrices for LSM readout using Stochastic Gradient Descent algorithm (% of predicted classes vs. % of actual class).

<i>Act. \ Pred.</i>	0	A	1	<i>Act. \ Pred.</i>	0	A	1	Total
0	37%	33%	31%	0	28%	42%	30%	100%
A	34%	35%	33%	A	26%	43%	31%	100%
1	30%	32%	36%	1	23%	41%	36%	100%
Total	100%	100%	100%					

The results for MLP were computed using the scikit-learn [180] package, based on several parameters¹⁰ listed in Table 4.21 and Adam optimiser [129].

A partial dependence graph presents the dependence of the predicted response on a single feature. The value of that dependence for a given class indicates by how much the log-probability for a class differs from the average class probability. As 10-fold cross-testing was applied, the partial dependence was computed on the full dataset. As it can be seen on Fig. 4.18a, the MLP readout is not particularly depended on *activation_value* feature.

The Accuracy calculated for the MLP-based LSM readout is 0.3618 (36%), while the F1 Score is 0.3548 (± 0.0092), that is *not a good result*. The MAUC for the MLP-based LSM readout is 0.536 (± 0.008), which is *not a good result*.

4.5.12 Detailed Results for LASSO-LARS

The LASSO-LARS is a scikit-learn [180] computed linear Lasso model fit with the Least Angle Regression (LARS). The algorithm, proposed in the 2003 by Bradley Efron et al. [64] computes so called Lasso path for all values of the regularisation parameter using LARS regression. It requires a number of passes on the data equal to the number of features. Its key parameters is *maximum_features*, that indicates the number of kept features. The value of 0 will enable all features, so will force no regularisation¹¹.

The Accuracy calculated for the LASSO-LARS LSM readout is 0.3551 (36%), while the F1 Score is 0.3528 (± 0.0042), that is *not a good result*. The MAUC for the same readout is 0.530 (± 0.0041), which is *not a good result*.

¹⁰Multi-layer Perceptron Documentation and Parameters https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html.

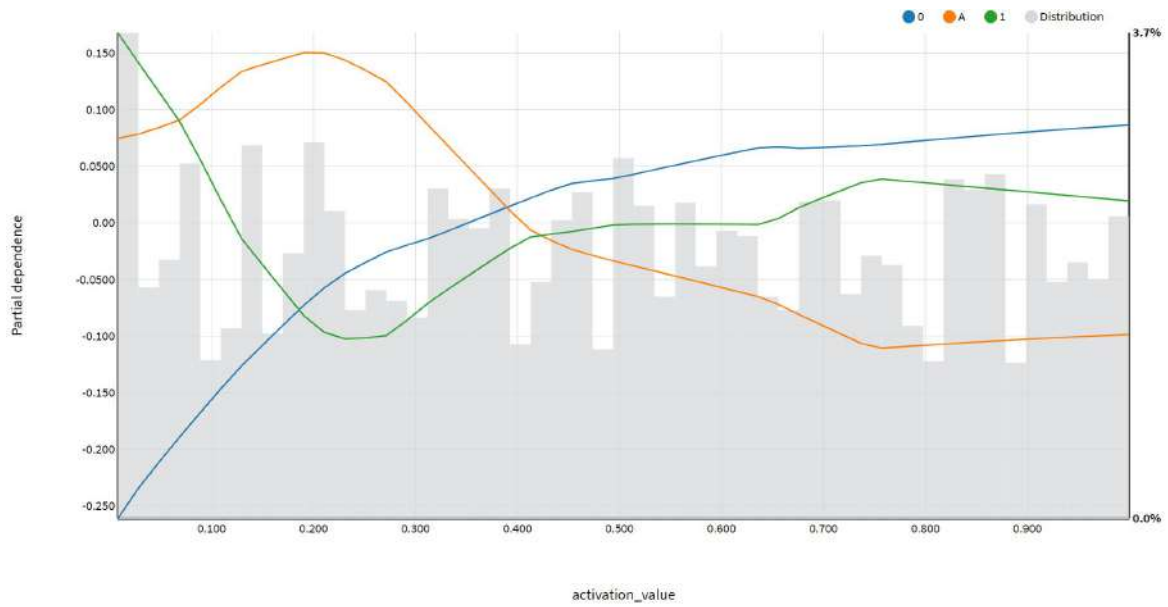
¹¹LASSO-LARS Documentation and Parameters https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LassoLars.html.

Table 4.21 Model parameters and detailed performance metrics for LSM readout using Multi-layer Perceptron.

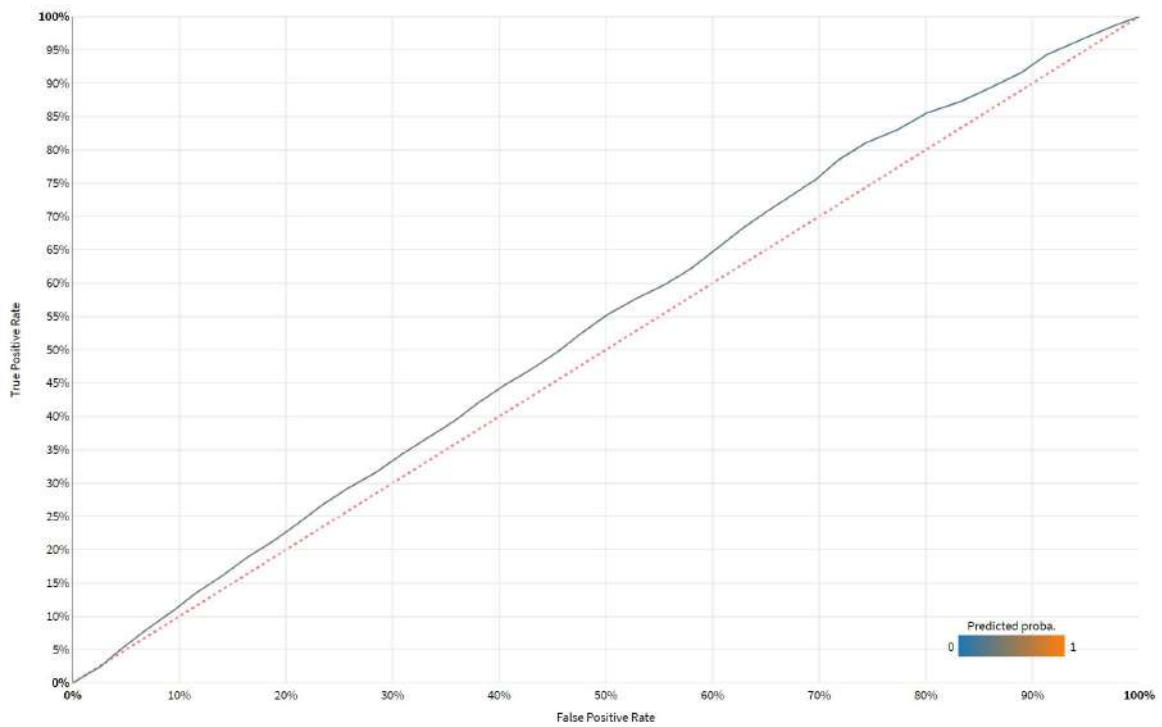
Parameter	Value	Metric	Value
Activation	ReLU	Precision	0.3625 (\pm 0.0103)
α	0.001	Accuracy	0.3618 (\pm 0.0079)
Max iterations	200	Recall	0.3618 (\pm 0.0079)
Convergence tolerance	0.0001	ROC - MAUC Score	0.5364 (\pm 0.0080)
Early stopping	No	F1 Score	0.3548 (\pm 0.0092)
Solver	Adam	Log loss	1.0947 (\pm 0.0018)
Shuffle data	Yes	Hamming loss	0.6382 (\pm 0.0079)
Initial Learning Rate	0.001		
Automatic batching	Yes		
Batch size	200		
Hidden layer sizes	100, 50, 25		
β_1	0.9		
β_2	0.999		
ϵ	1e-8		

Table 4.22 Confusion matrices for LSM readout using Multi-layer Perceptron (% of predicted classes vs. % of actual class).

<i>Act. \ Pred.</i>	0	A	1	<i>Act. \ Pred.</i>	0	A	1	Total
0	37%	32%	33%	0	23%	54%	23%	100%
A	33%	36%	29%	A	21%	60%	19%	100%
1	30%	31%	38%	1	20%	54%	27%	100%
Total	100%	100%	100%					



(a) Partial dependence graph for Multi-layer Perceptron readout. The figure presents 50 bins for *activation_value*, computed on the 10000 random sample.



(b) Multi-layer Perceptron Receiver Operating Characteristic curve. The AUC for class '0' is 0.536.

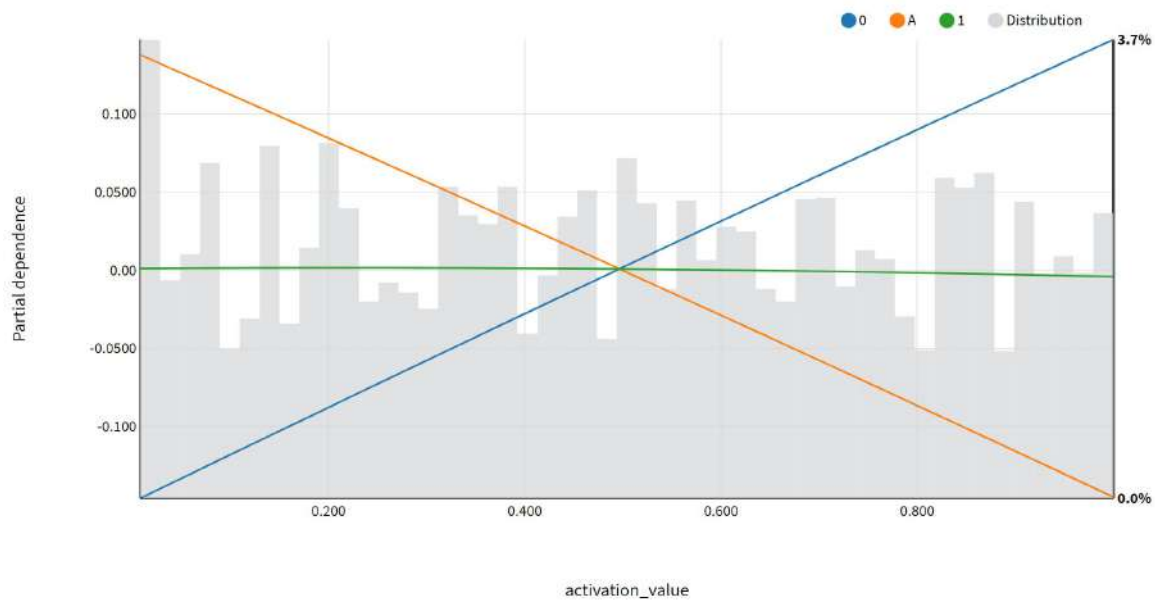
Fig. 4.18 Additional model details and performance for LSM readout using Multi-layer Perceptron.

Table 4.23 Model parameters and detailed performance metrics for LSM readout using LASSO-LARS algorithm.

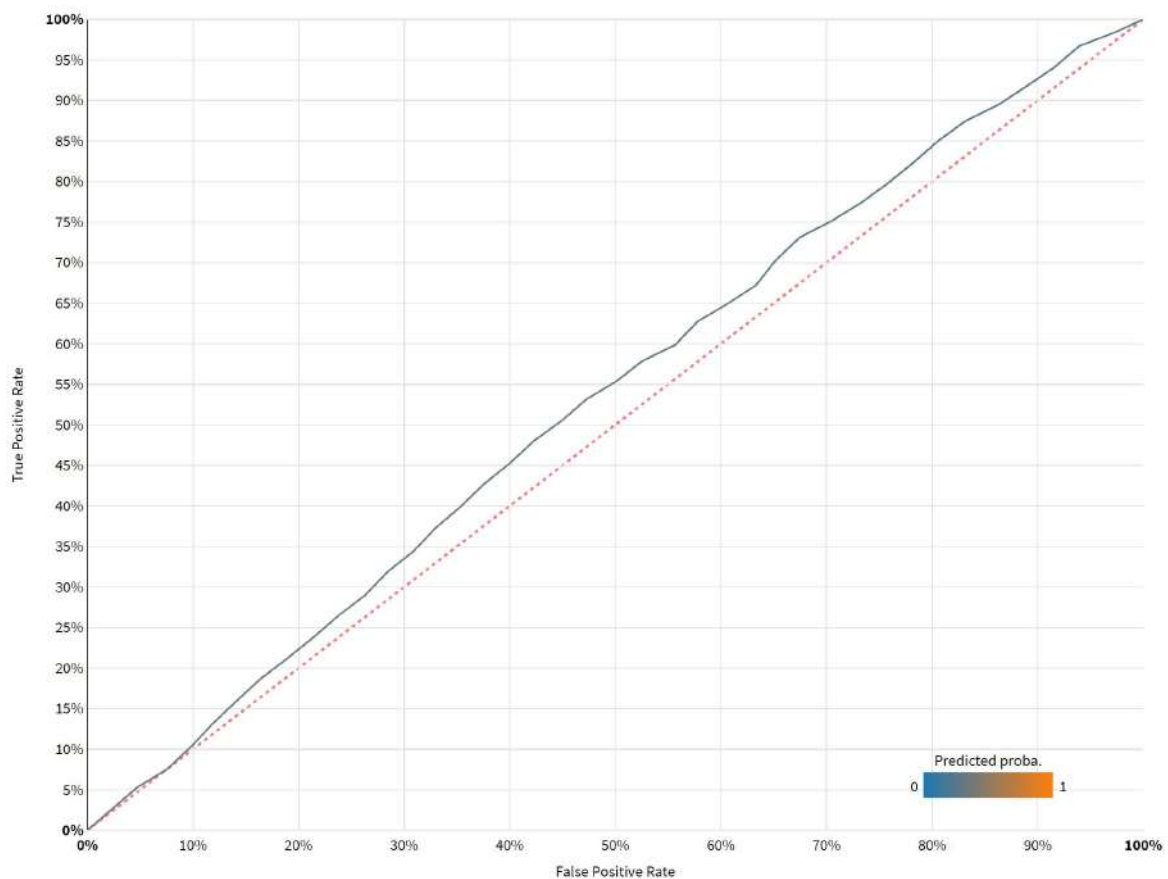
Parameter	Value	Metric	Value
Max number of features	0	Precision	0.3552 (\pm 0.0047)
		Accuracy	0.3551 (\pm 0.0043)
		Recall	0.3543 (\pm 0.0043)
		ROC - MAUC Score	0.5294 (\pm 0.0041)
		F1 Score	0.3528 (\pm 0.0042)
		Log loss	1.0960 (\pm 0.0007)
		Hamming loss	0.6449 (\pm 0.0043)

Table 4.24 Confusion matrices for LSM readout using LASSO-LARS algorithm (% of predicted classes vs. % of actual class).

<i>Act. \ Pred.</i>	0	A	1	<i>Act. \ Pred.</i>	0	A	1	Total
0	36%	33%	31%	0	29%	41%	30%	100%
A	34%	35%	33%	A	27%	42%	31%	100%
1	30%	32%	36%	1	24%	40%	35%	100%
Total	100%	100%	100%					



(a) Partial dependence graph for LASSO-LARS readout. The figure presents 50 bins for *activation_value*, computed on the 10000 random sample.



(b) LASSO-LARS Receiver Operating Characteristic curve. The AUC for class '0' is 0.534.

Fig. 4.19 Additional model details and performance for LSM readout using LASSO-LARS algorithm.

Table 4.25 Model parameters and detailed performance metrics for LSM readout using Support Vector Machine algorithm.

Parameter	Value	Metric	Value
Kernel	rbf	Precision	0.3591 (\pm 0.0064)
Kernel coef	scale	Accuracy	0.3564 (\pm 0.0050)
C	1	Recall	0.3580 (\pm 0.0047)
Stopping tolerance	0.001	ROC - MAUC Score	0.5330 (\pm 0.0049)
Max iterations	-1	F1 Score	0.3489 (\pm 0.0058)
		Log loss	1.0955 (\pm 0.0009)
		Hamming loss	0.6436 (\pm 0.0050)

Table 4.26 Confusion matrices for LSM readout using Support Vector Machine algorithm (% of predicted classes vs. % of actual class).

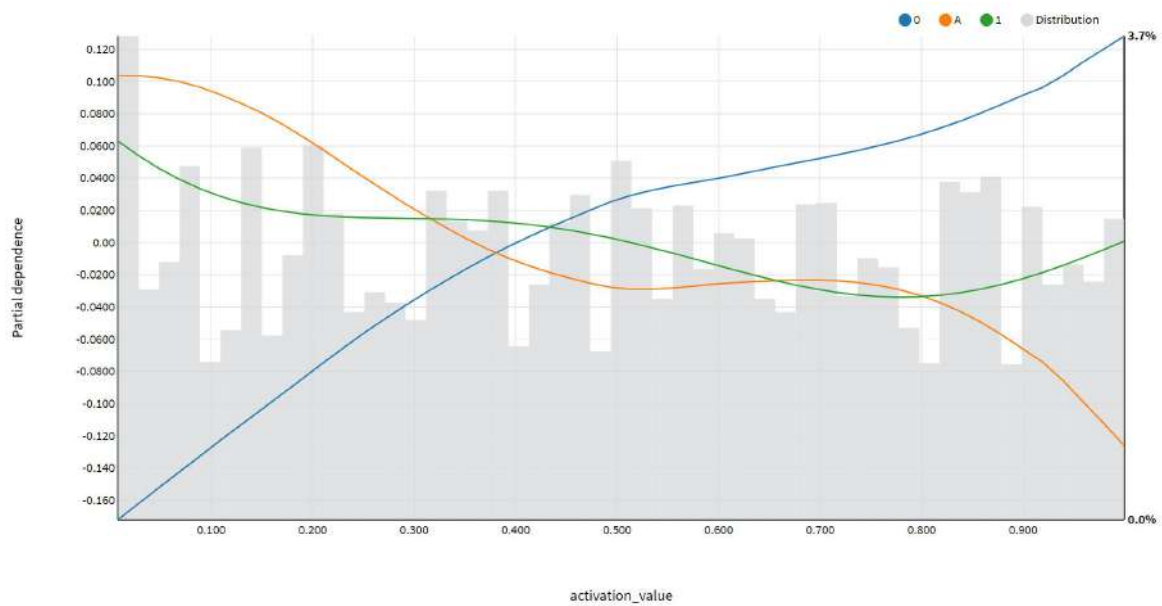
<i>Act. \ Pred.</i>	0	A	1	<i>Act. \ Pred.</i>	0	A	1	Total
0	36%	31%	31%	0	53%	19%	28%	100%
A	33%	37%	32%	A	49%	22%	29%	100%
1	31%	32%	36%	1	47%	20%	33%	100%
Total	100%	100%	100%					

4.5.13 Detailed Results for Support Vector Machines

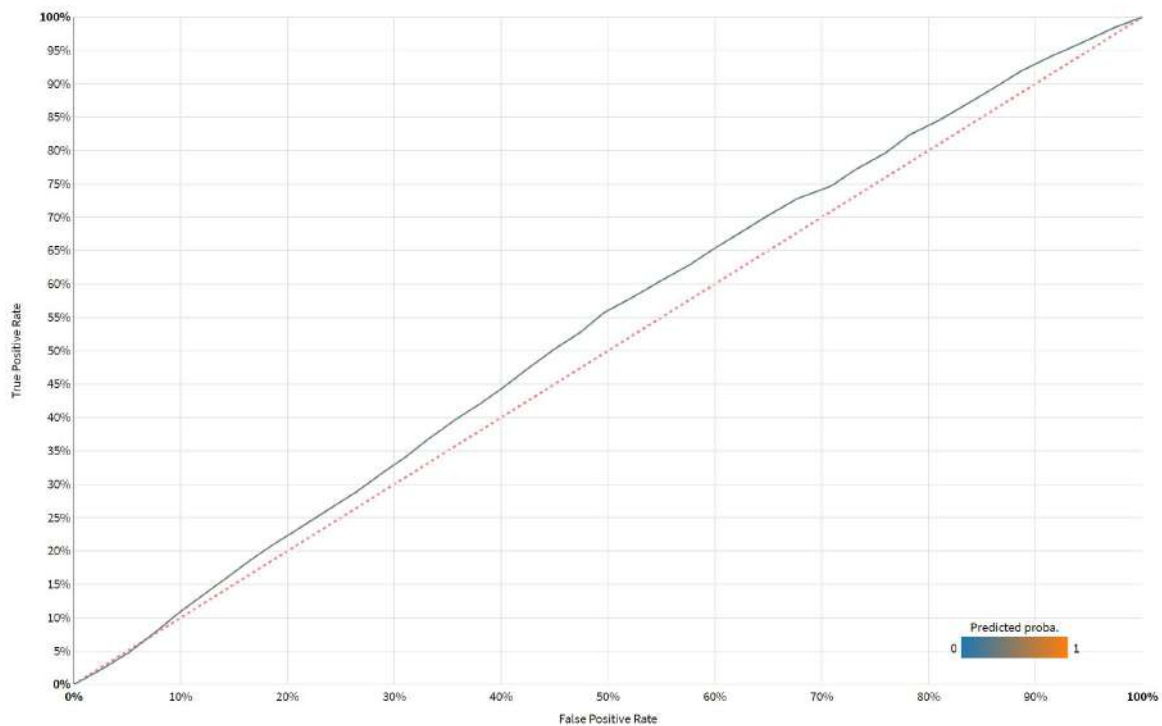
Support Vector Machines (SVM) are a type of supervised learning algorithms that can be used for classification or regression tasks. The goal of an SVM is to find the best possible hyperplane that can linearly separate a dataset into different classes. The hyperplane is chosen so that it maximises the margin, or distance, between the nearest data points of different classes. Once the hyperplane is chosen, new data points can be easily classified by determining on which side of the hyperplane they fall. SVMs are universal, effective in high-dimensional spaces and often used for text or image classification [190]. The calculation of SVM LSM readout was performed with scikit-learn [180] package, using a number of pre-selected parameters¹².

The Accuracy calculated for the Support Vector Machines LSM readout is 0.3564 (36%), while the F1 Score is 0.3589 (\pm 0.0058), that is *not a good result*. The MAUC for the same readout is 0.5330 (\pm 0.0049), which is *not a good result*.

¹²Support Vector Machines Documentation and Parameters <https://scikit-learn.org/stable/modules/svm.html#support-vector-machine>.



(a) Partial dependence graph for Support Vector Machine readout. The figure presents 50 bins for *activation_value*, computed on the 10000 random sample.



(b) Support Vector Machine Receiver Operating Characteristic curve. The AUC for class '0' is 0.533.

Fig. 4.20 Additional model details and performance for LSM readout using Support Vector Machine algorithm.

Table 4.27 Model parameters and detailed performance metrics for LSM readout using Decision Tree classifier.

Parameter	Value	Metric	Value
Max tree depth	5	Precision	0.4649 (\pm 0.0264)
Split criterion	gini	Accuracy	0.3819 (\pm 0.0070)
Min samples per leaf	1	Recall	0.3813 (\pm 0.0058)
Splitter	best	ROC - MAUC Score	0.5590 (\pm 0.0051)
		F1 Score	0.3283 (\pm 0.0090)
		Log loss	1.0620 (\pm 0.0050)
		Hamming loss	0.6181 (\pm 0.0070)

4.5.14 Detailed Results for Decision Tree

A Decision Tree (DT) classifier works by constructing a tree-like model of decisions and their possible consequences, including the predicted outcome for each possible decision. The decision tree algorithm splits the training data into subsets based on the most important attributes, and uses these splits to make predictions about the target variable for new data. The calculation of the DT LSM readout was performed with scikit-learn [180] package.

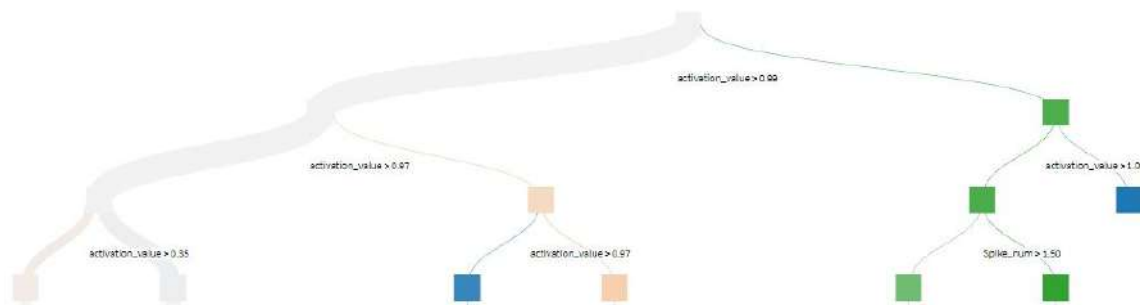
The key parameters of a DT classifier [26] include the criterion used to split the data, the maximum depth of the tree, the minimum number of samples required to split a node, and the minimum number of samples required to be at a leaf node. The criterion determines how the algorithm selects the most important attributes for each split, and can be either the Gini index or the entropy. The maximum depth of the tree controls the complexity of the model, and can be used to prevent overfitting. The minimum number of samples required to split a node and the minimum number of samples required to be at a leaf node are used to control the amount of data used to build the model, and can also help prevent overfitting. These parameters can be tuned to improve the performance of the model on unseen data¹³.

The Accuracy calculated for the DT LSM readout is 0.3819 (38%), while the F1 Score is 0.3283 (\pm 0.0090), that is *not a good result*. The MAUC for the same readout is 0.5590 (\pm 0.0051), which is *not a good result*.

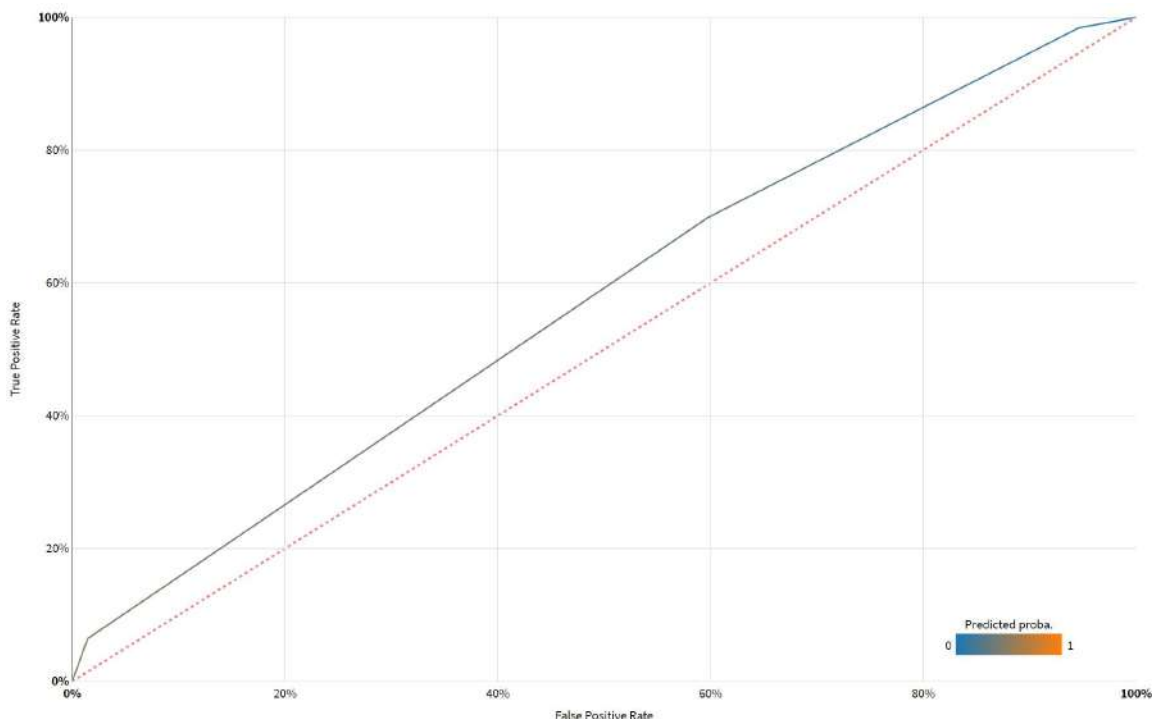
¹³Decision Trees Documentation and Parameters <https://scikit-learn.org/stable/modules/tree.html#decision-trees>.

Table 4.28 Confusion matrices for LSM readout using Decision Tree classifier (% of predicted classes vs. % of actual class).

<i>Act. \ Pred.</i>	0	A	1	<i>Act. \ Pred.</i>	0	A	1	Total
0	37%	28%	24%	0	68%	30%	2%	100%
A	31%	40%	12%	A	57%	42%	1%	100%
1	32%	31%	64%	1	60%	34%	7%	100%
Total	100%	100%	100%					



(a) Sample Decision Tree readout. The rightmost leaf presents the prediction of target classes: '0' (100%), 'A' and '1' (0%) using the decision rule *activation_value* > 0.99. Samples count 19 (0.01%).



(b) Decision Tree Receiver Operating Characteristic curve. The AUC for class '0' is 0.570.

Fig. 4.21 Additional model details and performance for LSM readout using Decision Tree algorithm.

Table 4.29 Model parameters and detailed performance metrics for LSM readout using AdaBoost algorithm.

Parameter	Value	Metric	Value
Estimators	20	Precision	0.3715 (\pm 0.0061)
		Accuracy	0.3708 (\pm 0.0052)
		Recall	0.3703 (\pm 0.0056)
		ROC - MAUC Score	0.5456 (\pm 0.0092)
		F1 Score	0.3692 (\pm 0.0059)
		Log loss	1.0979 (\pm 0.0003)
		Hamming loss	0.6292 (\pm 0.0052)

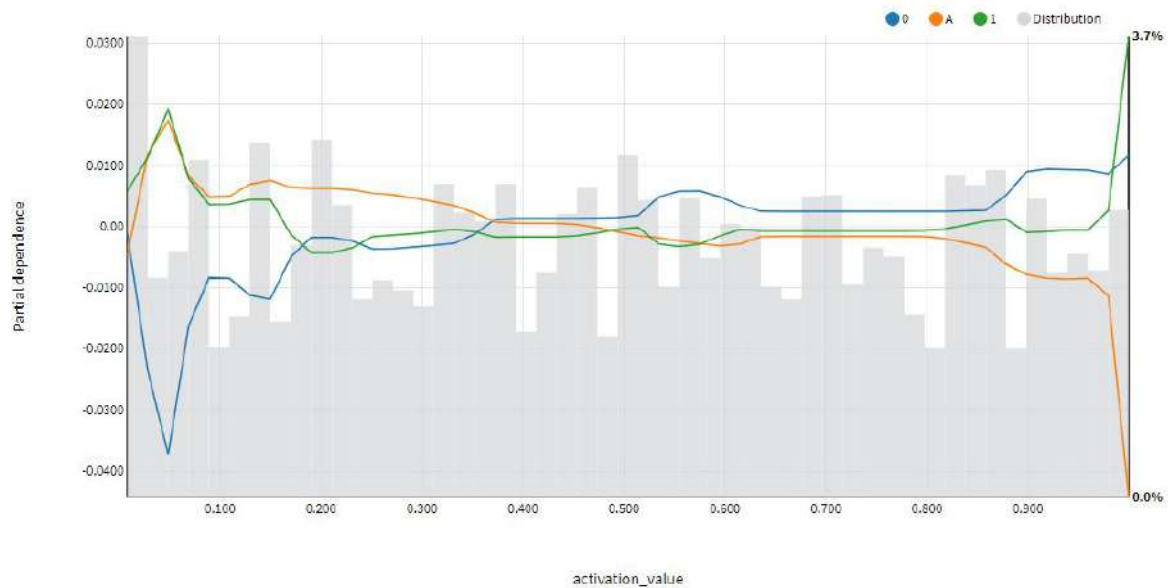
Table 4.30 Confusion matrices for LSM readout using AdaBoost algorithm (% of predicted classes vs. % of actual class).

<i>Act.</i> \ <i>Pred.</i>	0	A	1	<i>Act.</i> \ <i>Pred.</i>	0	A	1	Total
0	37%	32%	31%	0	35%	40%	28%	100%
A	33%	37%	30%	A	31%	45%	29%	100%
1	30%	31%	39%	1	27%	40%	33%	100%
Total	100%	100%	100%					

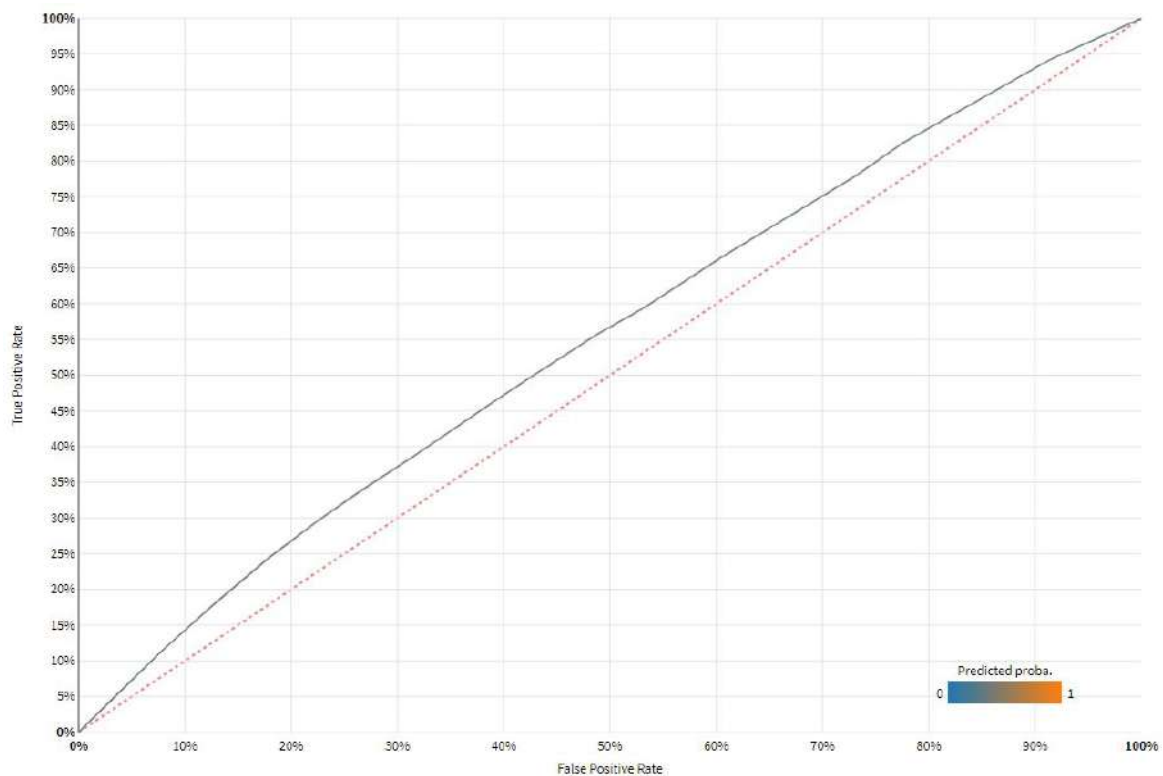
4.5.15 Detailed Results for AdaBoost Classifier

AdaBoost, short for “Adaptive Boosting” is a type of ensemble learning algorithm that can be used for classification or regression tasks [72]. It combines multiple weak learners to create a strong learner that can make accurate predictions. A weak learner is a model that is only slightly better than random guessing, while a strong learner is a model that has a low error rate. AdaBoost works by weighting the training data points and iteratively training the weak learners on different subsets of the data. The weak learners are then combined to form a final strong learner, which makes predictions by taking a weighted majority vote of the weak learners. This LSM readout uses the scikit-learn [180] implementation known as AdaBoost-SAMME [95] with 20 estimators.

The Accuracy calculated for the AdaBoost LSM readout is 0.3708 (37%), while the F1 Score is 0.3692 (\pm 0.0059), that is *not a good result*. The MAUC for AdaBoost LSM readout is 0.546 (\pm 0.009), which is *not a good result*.



(a) Partial dependence graph for AdaBoost readout. The figure presents 50 bins for *activation_value*, computed on the 10000 random sample.



(b) AdaBoost Receiver Operating Characteristic curve. The AUC for class '0' is 0.552.

Fig. 4.22 Additional model details and performance for LSM readout using AdaBoost algorithm.

4.6 Experimental Summary

In this experimental chapter, author focused, among some other introductory experiments, the performance of twelve machine learning algorithms used in the training of own LSM model RetNet(5x8,1) for the classification task differentiating 3 input patterns consisting of the numbers (“0”, “1”) and a letter (“A”). The algorithms evaluated in detail include LightGBM, Gradient Boosted Trees, XGBoost, Extra Trees, Random Forest, Logistic Regression, SGD, Multi-layer Perceptron, LASSO-LARS, SVM, Decision Tree, and AdaBoost.

The dataset with labelled samples was used, and as described in Subsection 4.5.1, a 10-fold cross-validation was performed to evaluate the classification accuracy, precision, recall, F1 score, ROC AUC, as well as the training time of each algorithm.

The results showed that the best performing algorithms in terms of accuracy were LightGBM (81%), with Gradient Boosted Trees, and XGBoost, all of which achieved over 67% accuracy. Random Forest and Extra Trees also performed relatively well, achieving accuracies of around 50%. Logistic Regression, SGD, Multi-layer Perceptron, LASSO-LARS, SVM, and Decision Tree had lower accuracy, ranging from 36% to 38%, with the SVM, basic Decision Trees and AdaBoost being the worst performers.

In terms of training time, the fastest algorithms were Decision Trees, SGD, and Logistic Regression (less than 1 minute), while the slowest were SVM (44 hours), LightGBM (18 minutes), and XGBoost (11 minutes).

Overall, the results suggest that LightGBM, Gradient Boosted Trees, and XGBoost are good choices for this LSM classification task, with LightGBM being the most accurate (81% and 18 minutes of training), while Gradient Boosted Trees being also accurate, and relatively fast (68% and a three times smaller training time of 6 minutes).

Chapter 5

Neural Simulation Pipeline

5.1 Introduction

In the fifth chapter of this PhD dissertation the author describes *the details of Neural Simulation Pipeline (NSP)*. The chapter explains the purpose and architecture of the NSP, as well as its limitations. We start with a description of why prototyping with Single Board Computer Clusters is a good idea, and we introduce some details about the Neural Simulation Cluster (NSC). We continue with the introduction of Neural Simulation Pipeline that runs our selected simulation engine (GENESIS). We follow with a characterisation of the pipeline, its components, architecture, scripts, as well as the different aspects of Docker containerisation. Finally, author concludes with a description of NSP's limitations and general challenges.

The chapter is organised as follows. This Section 5.1 provides a discussion of this chapter. Section 5.2 explains why prototyping with Single Board Computer Clusters is important for neural computations, whereas Section 5.3 summarises how Neural Simulations Cluster was built. Section 5.4 introduces Neural Simulation Pipeline. Subsection 5.4.1 describes the components of Neural Simulation Pipeline. Subsection 5.4.2 explains the architecture of the pipeline, whereas Subsection 5.4.3 focuses on the containerisation with Docker. Subsection 5.4.4 presents key elements of the Docker architecture. Subsection 5.4.5 describes kernel internals, while Subsection 5.4.6 focuses on Docker Engine and Neural Simulation Pipeline's scripts. Finally, Section 5.5 describes the results of experimental evaluation of Neural Simulation Pipeline, while Section 5.6 highlights the key limitations of the solution.

Overall, this chapter presents the details about Neural Simulation Pipeline and Neural Simulations Cluster, that there proposed to lower the entry barriers to the simulation of brain networks.

5.2 Prototyping with Single Board Computer Clusters

Single Board Computer Clusters (SBCC) are increasingly popular, as they provide an inexpensive alternative to traditional large scale HPC clusters. The large systems are often in high demand, and it's often difficult to book cluster time. This new class of tiny, low-cost computers are used in a number of ways. As newer and more capable boards are frequently released, these SBCCs are now used not only to build clusters for educational purposes, but also as a lower cost test environments for the new data centre technologies e.g. high core-density architectures [124].

These clusters are becoming increasingly popular and powerful. In 2018 Linux.com [34] reviewed the new projects, and concluded that on most SBC clusters are between 5 to 25 boards. A typical project of this type, as mentioned in the SBCC review [34] was Balena's Beast v2¹. It is a testing and demo rig built of 144 RPi nodes. The system's weight is 150 kg, measuring almost 2 meters.

The most important SBCC projects worldwide was deployed to Los Alamos National Laboratory. It was based on 750 RPi nodes encapsulated into five rack-mount cluster modules², each containing 150 quad-core ARM processors. This gave Los Alamos HPC software developers access to a 3,000-core cluster³ for a fraction of cost needed to build a proper petascale development environment. The alternative would cost a quarter billion dollars and use 25 megawatts of electricity, so the machine is now increasing to 4,000-cores.

Although research centres and universities around the world have developed SBCC for research into parallel computing, deep learning, medical research, weather simulations, crypto-currencies mining, software-defined networks, distributed storage, redundancy or to simulate massive Internet of Things (IoT) networks for a couple of years now, the interest on the commercial IT vendors' side was a bit smaller.

This is changing, as since 2019 even leading IT vendors like Oracle started using and promoting SBC clusters. A few projects from Oracle touched upon building so called multi-purpose "supercomputers" [168]. A good example of that was *The Oracle Raspberry Pi Supercomputer Project*⁴, built of 1060 RPi SBCs running Oracle Linux and Java to "help with prototyping, and open embedded computing to a wide audience". The machine was built of 2 rack units (RU) containing 21 Raspberry Pi 3B+ nodes each, that had their carriers

¹Balena's Beast RPi Project <https://www.balena.io/blog/the-evolution-of-the-beast-continues/>

²Raspberry Pi Newsroom <https://www.raspberrypi.com/news/raspberry-pi-clusters-come-of-age/>

³Los Alamos RPi Supercomputing Project <https://top500.org/news/lanl-turns-to-raspberry-pi-for-supercomputing-solution/>

⁴Oracle Raspberry Pi Supercomputer <https://www.servethehome.com/oracle-shows-1060-raspberry-pi-supercomputer-at-oow/>

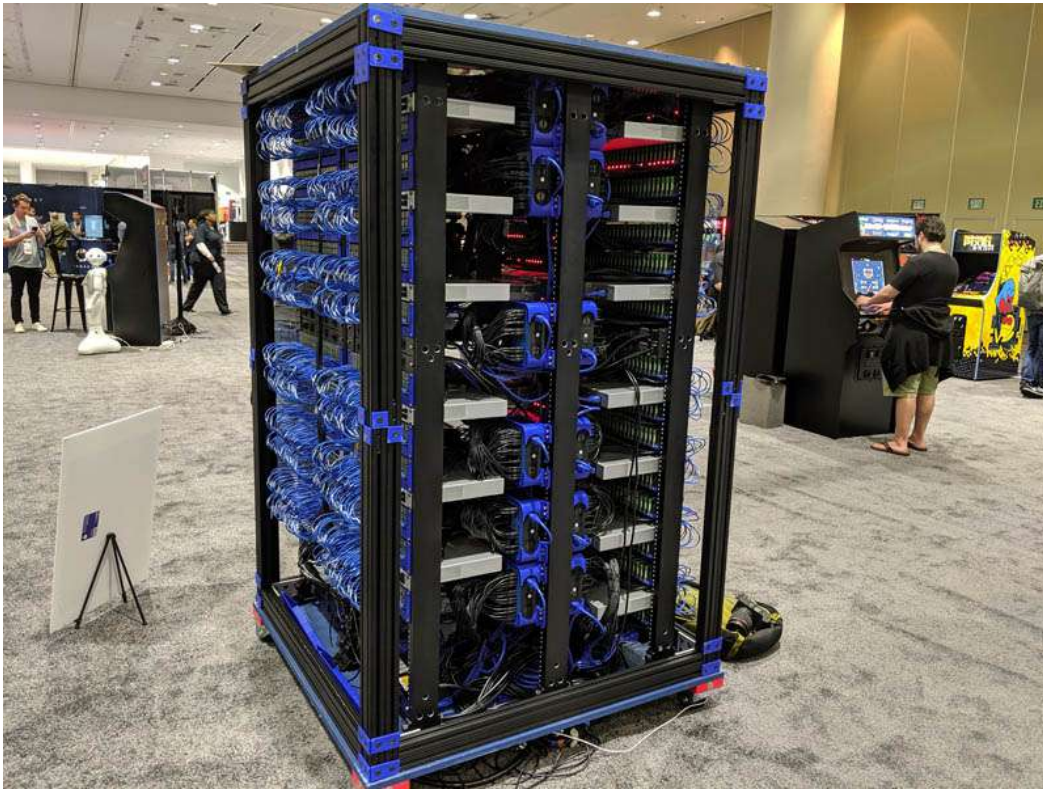
3D printed. Each individual Oracle cluster node was connected through the 48-port Ubiquiti UniFi switches with SFP+ 10 Gb Ethernet cables. Both clusters are presented in Fig. 5.1

However, in any recent SBCC reviews author has not found an indication of any such project being specifically focused on simulating cybernetic models; neither in the area of research, nor even in teaching neuroinformatics, biomedical engineering or computational neuroscience. This might be because the scientific community as a whole still does not really consider SBCC to be a practical tool for performing computer simulations due to its low compute performance offered. This notion is changing. In 2020 Bastford et al. [13] showed that the performance improvements of SBC mean that “for the first time SBC clusters have moved from being a curiosity to a potentially useful technology”.

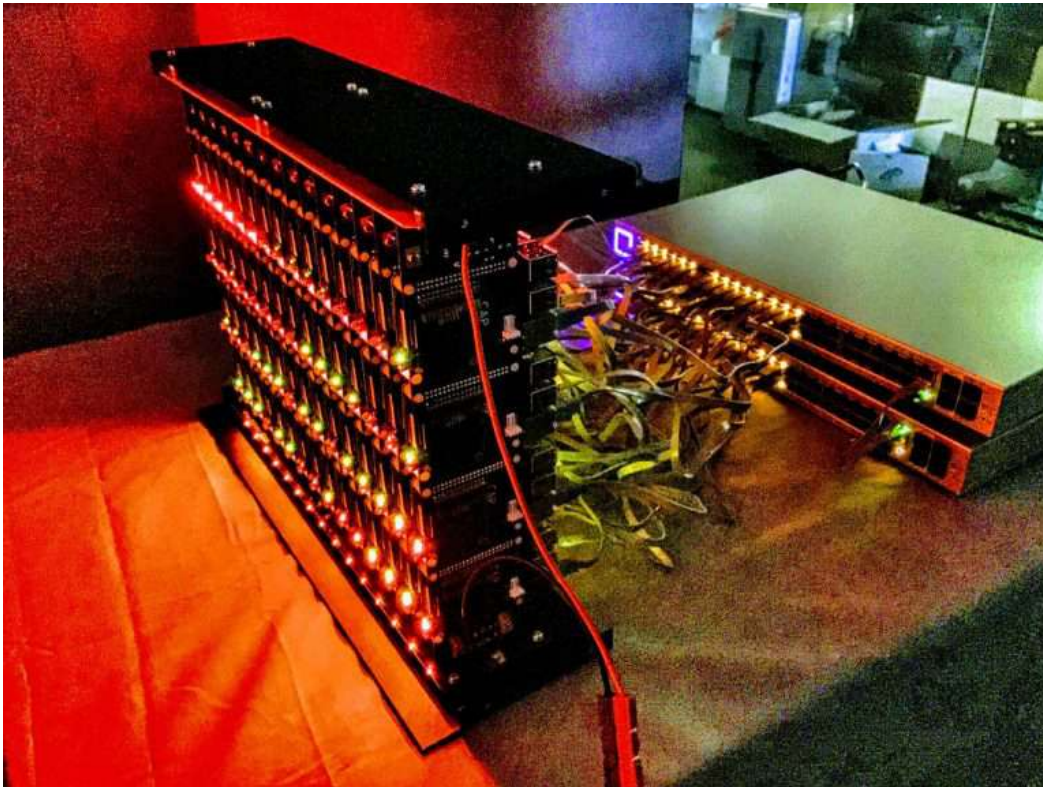
These SBCC systems typically use the same MPI library [77] that is installed in HPC environments. The library is used as an interface for exchanging messages between computing nodes to deploy a parallel programs across distributed memory. Although SBC clusters are not as computationally powerful as HPC clusters, author believes that these systems can still play an important role in neural simulations. As described in Chapter 1.4 and 2.10 neural simulations require not only the top performance, but also the scale provided by a combination of separate cores running in parallel.

As mentioned by Johnston et. al [124], in the era of IoT the computer architectures “tend to move away from a centralised compute resource, to an architecture where the computational power is pushed out closer to the edge”. Authors suggest that the two main advantages of the SBCC in this environment are (1) their low power consumption and (2) size. In addition to that, these SBCCs can be flexibly powered on e.g. only when certain computational resources are needed, so that they can both be energy efficient, and cope with peak computational demands resulting e.g. from bursting of audio, video or seismic data processing based on trigger events. The authors foresee new, even more powerful and specialised SBCCs to appear in the upcoming years. There are four key reasons why a specialised cluster, built of SBCs could be useful:

- Cost-effectiveness - SBCs are generally much cheaper than traditional computers, making it more cost-effective to build such a cluster of them.
- Portability - SBCs are small and lightweight, which makes them easy to transport and set up in different locations. This could be very useful in dialectics and for demonstrations, as such a cluster can easily be used in a classroom setting.
- Power efficiency - SBCs are also known for their low power consumption, which makes them more energy-efficient than traditional computers. This is important for neural simulation, which can be computationally intensive and require a lot of power.



(a) Oracle Raspberry Pi Supercomputer built of 1060 Raspberry Pi 3B+ computers.



(b) A single node of Los Alamos National Laboratory's 3000 Raspberry Pi cluster.

Fig. 5.1 Two exemplary SBCC Cluster projects: HPC development environment for Los Alamos National Laboratory, as well as an Oracle Raspberry Pi Supercomputer presented at Oracle OpenWorld in 2019 [34].

- Scalability - It is easy to add more SBCs to the cluster as the need for more computational power grows. This allows the cluster to scale up as needed, making it more flexible and adaptable.

Overall, the author of this PhD thesis thinks that building a specialised SBC cluster focused on neural simulation can be a cost-effective, energy-efficient, portable, and scalable way to perform simulation tasks.

5.3 Neural Simulation Cluster

As mentioned in the previous Section 5.2, there are a few reasons why building a specialised cluster of SBCs could be useful in designing, developing and prototyping large neural simulations of brain networks. This section introduces the key design considerations and system components of the specialised SBCC called *Neural Simulations Cluster (NSC)*.

When designing the NSC for a home environment, author considered several factors such as power consumption, space, access to internet and networking, cooling, noise, and overall cost to ensure that such a cluster could be built with a limited budget, and used effectively in author's home environment. The design had to be simple and flexible. Fig. 5.2 presents the result of these considerations, a level design diagram of this machine, that was used for all the prototyping needed for this PhD thesis in author's home environment. The cluster was also used to execute and evaluate Neural Simulation Pipeline (NSP).

All the technical details related to building the NSC have been gathered in the Appendix B of this thesis. The author selected a single four-core Raspberry Pi 4 Model B board, marked in blue on the Fig. 5.2 (a full specification of the board in Appendix B.1), and the three hexa-Core ROCKPro64 boards, marked in green on the Fig. 5.2 (a full specification of the board in Appendix B.2), both based on the same ARM64 architecture. All boards of the cluster have been connected using a Gigabit Ethernet switch. The SBC boards were selected and procured not without challenges, as described in Appendix "Building Neural Simulations Cluster" B.1.

The RPi board was selected as a control node of the cluster, due to its out-of-the box connectivity using a DualBand WiFi (2,4 GHz and 5 GHz), as well as the Bluetooth 5. Moreover, in author's view the quad-core Broadcom BCM2711 ARM-8 Cortex-A72 CPU running at 1.5 GHz offers a satisfactory performance for prototyping parallel situations.

On the other hand, the three RPr boards, each equipped with a Rockchip RK3399 (quad-core ARM Cortex A53 CPU running at 1.4 GHz, and dual-core ARM-8 Cortex-A72 running at 1.8 GHz), offer 18 additional cores to enhance performance of prototyping more complex (e.g. distributed) simulations. Nevertheless, as per the initial tests in author's home

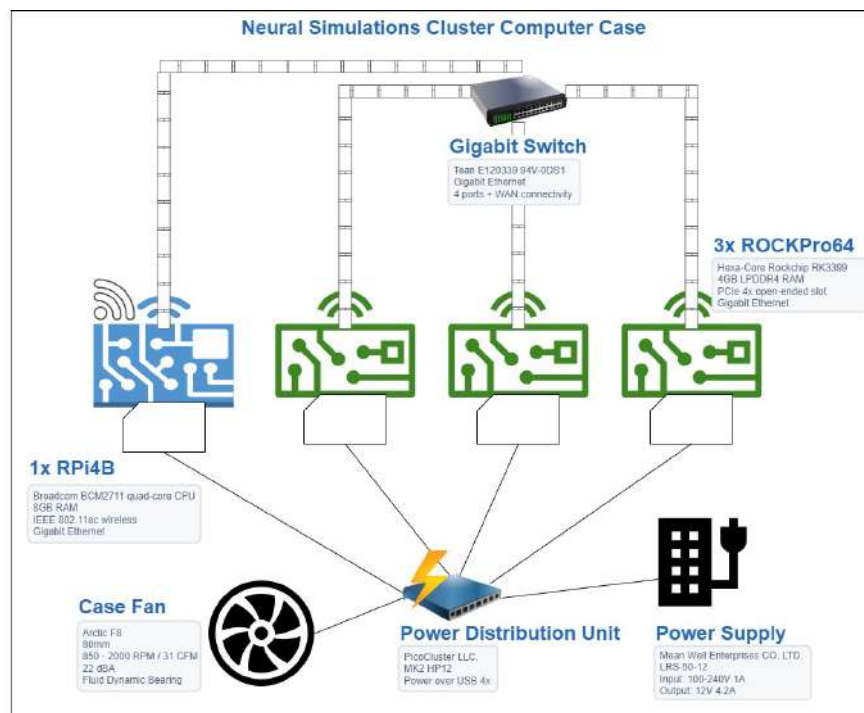


Fig. 5.2 High Level Design Diagram for Author's *Neural Simulations Cluster*.

environment, this additional computing power is available not without its disadvantages. The RPr board generate noticeably more heat, and in contrast to the RPi, they do not run in a stable way without an active cooling system.

The result configuration of NSC is therefore a hybrid of “a silent” RPi, perfect for author's home environment, and “a noisy” (due to a case fan) ROCKPro configuration, that could however be switched off easily when not needed for any simulations. This aspect impacted the selection of the cluster case (PicoCluster), and its modification for the RPi, as presented in Fig. 5.4. Author assembled all the hardware elements from the SBC kit described in Appendix B.1.1. The complete, working NSC system is presented in Fig. 5.3.

As per the bill of materials presented in Appendix B.1, the total cost of NSC was PLN 4492.44 only. The configuration proposed allows for prototyping both the parallel, and distributed simulations.

The system uses five *NSC scripts* that facilitate some standard operations on the cluster, all gathered below in this Subsection 5.3.

Neural Simulation Cluster Scripts

1. *testNscNodes.sh* - Bash script that performs the *uptime* and *df -h* commands on each node of the cluster to validate if that node is up and running.



(a) Neural Simulation Cluster, room light on.



(b) Neural Simulation Cluster, room light off.

Fig. 5.3 Photos of the assembled Neural Simulations Cluster in author's home environment.

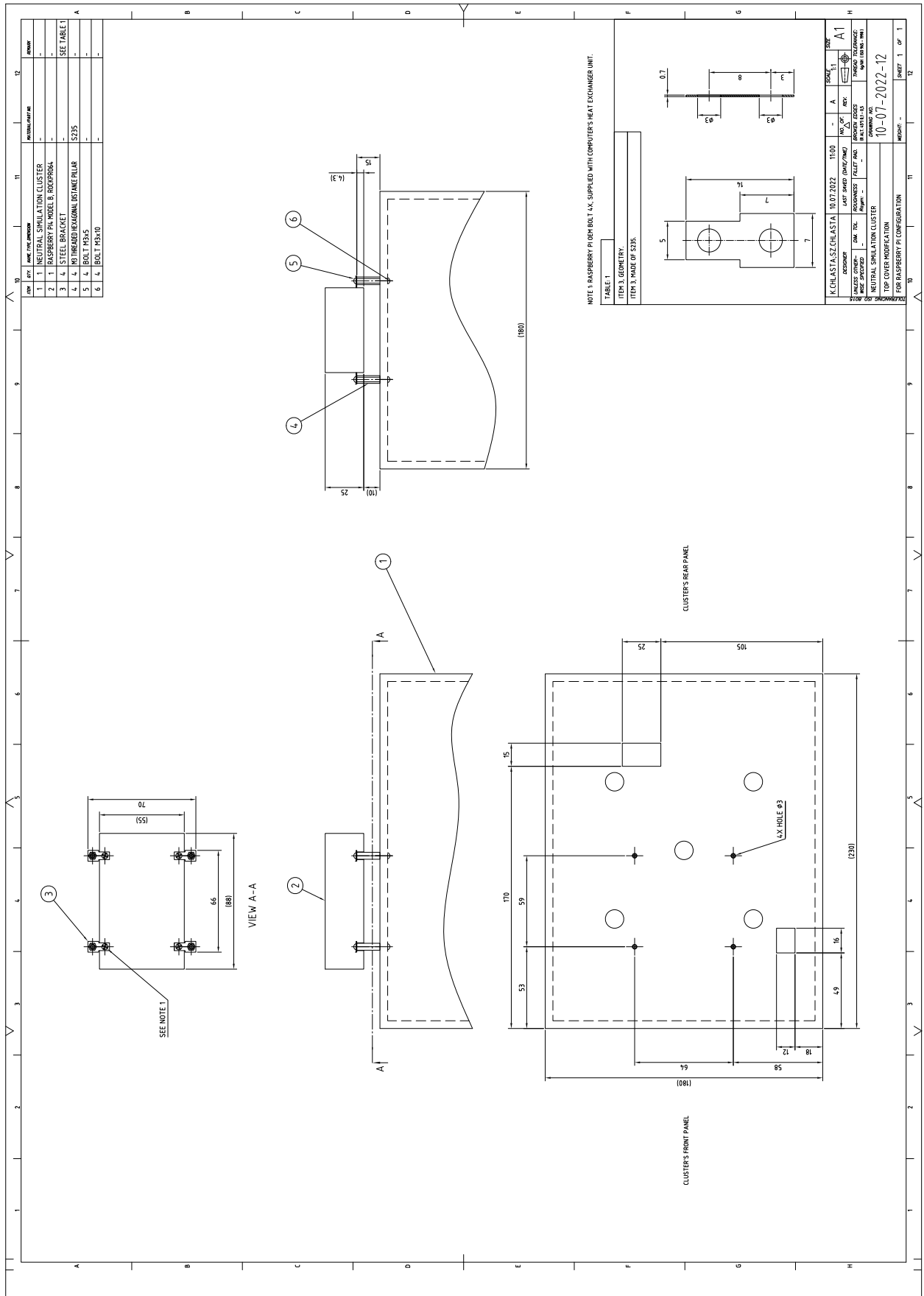


Fig. 5.4 Technical drawing of Neural Simulation Cluster's top cover modification.

- *Inputs:* N/A
 - *Outputs:*
 - (a) Node's up-time, number of users, and system load summary sent to standard output.
 - (b) Node's amount of disk space available on the file system sent to standard output.
2. *stopNscNodes.sh* - Bash script that shuts down all of the NSC nodes. The script needs to be run before turning off and unplugging the cluster to avoid a high risk of corrupting the Micro SD cards.
- *Inputs:* N/A.
 - *Outputs:*
 - (a) NSC is ready to be switched off (using the power switch).
3. *restartNscNodes.sh* - Bash script that restarts all the nodes of NSC.
- *Inputs:* N/A.
 - *Outputs:*
 - (a) NSC restarted.
4. *resizeNscNodes.sh* - Bash script that resizes the root file system of the NSC node to be of a size matching the desired micro SD card. This allows using standard operating system and application images. The underlying *resizeNscFs.sh* script needs *sudo* privileges to execute; it's based on the best practice provided by the supplier of cluster kit [33].
- *Inputs:*
 - (a) Desired root file system size.
 - *Outputs:*
 - (a) Root file system resized.
5. *genNscKeys.sh* - Bash script that generates the SSH key for the control node. This key has to be copied to all other nodes in the cluster, that are added to the *known_hosts* file.
- *Inputs:* N/A.
 - *Outputs:*

- (a) NSC's SSH identification is generated and saved to the home directory in `/.ssh/id_rsa`.
- (b) NSC's SSH public key is generated and saved to the home directory in `/.ssh/id_rsa.pub`.

5.4 Neural Simulations Pipeline

5.4.1 Components

As already mentioned building, testing and deploying cybernetic models often requires different software libraries as dependencies, and the ability to consume significant amounts of parallel or distributed computing power to speed up the long running computations. This is especially visible when developing larger, and more complex scripts that research human brain elements through long running simulations.

Author believes that the resolution of problem of building, testing and deploying cybernetic models, as well as their different software dependencies could be facilitated by using Docker [162]. Docker, that is described in Sections 5.4.3, 5.4.4, 5.4.5 and 5.4.6 is the most popular container platform; as suggested in the recent IDC's white paper [40], it has already attracted "a significant amount of industry recognition", and it has the opportunity to "define the road map for all container platforms". It also has a potential to become a key component of "all the enterprise IT environments globally", due to its operating system neutrality. The same report mentions that "the container revolution is under way", with a forecast of 1.8 billion enterprise container instances deployed by the end of 2021.

The complexity not only creates entry barriers for people, who want to start with brain simulations, but it can also impact smaller research or clinical centres working on brain research, that may be challenged by the technical issues and multiple resources required to implement the suitable pipelines in practice. This thesis presents a case study on how such a framework, using GENESIS simulator can be applied to brain simulations as a Neural Simulations Pipeline (NSP). The subsection also discusses the cloud-based elements, automation, testing and deployment resources available to researchers.

One of the practices used in modern software development is CI/CD. This approach focuses on automating the development cycle to reduce the likelihood of accidental human error in the process, as well as to save time through [199]:

- Enforcing the preparation of tests at multiple levels.
- Enforcing a regular execution of these tests to give developers, as well as the wider team *an immediate feedback* on the progress.

- Automating the testing through a standard process.
- Automating the deployment to production itself.

As a result, in a common scenario, thanks to the ability of seeing that a feature that has just been added breaks the whole build process, the bug can be fixed immediately, without bringing any new risk to the subsequent elements of the development cycle. If CI/CD is successfully used by the development teams, and a pipeline to automate the process is fully introduced, both time to market and human interactions can be reduced.

The following sections will introduce the architecture of Neural Simulation Pipeline (NSP), that automates some repetitive tasks in developing cybernetic modes and setting up experiments. The pipeline was developed with Bash [185].

NSP manages experiments and allows them to be saved and defined for different simulation engines in a unified way. The framework allows for the queue of experiments to be managed centrally, regardless of which hardware platform they are running on, in particular whether it is a Docker container in the cloud, or running on-premise. This approach also enables faster analysis of experimental data, as (1) all the experiment results are centrally stored and (2) the experiment data is partially pre-processed for further analysis, by aggregating the results together with the statistics on the execution environment (e.g. experiment run-times, detailed information about CPUs, memory and operating system processes).

5.4.2 Architecture

The high level design diagram of Neural Simulation Pipeline is presented in Figure 5.5. Different components of NSP connect to the AWS cloud via the AWS CLI (AWS Command Line Interface); sending requests to the AWS services by using HTTPS on TCP port 443. For security reasons we propose to create at least two types of user accounts with AWS IAM Service: an account for *a user persona*, focused on simulation design and execution (the data), and a separate one for *a developer and maintainer persona*, focused on the development of simulation models and administration of the pipeline through the code. We propose that these two different types of personas interact with the system via different interfaces:

- *A user persona*, via *NSP scripts* described in Table 5.1.
- *A developer and maintainer persona*, via Git client (code), AWS CLI and AWS Management Console (actions)⁵.

⁵AWS Cloud Products website <https://aws.amazon.com/products/>

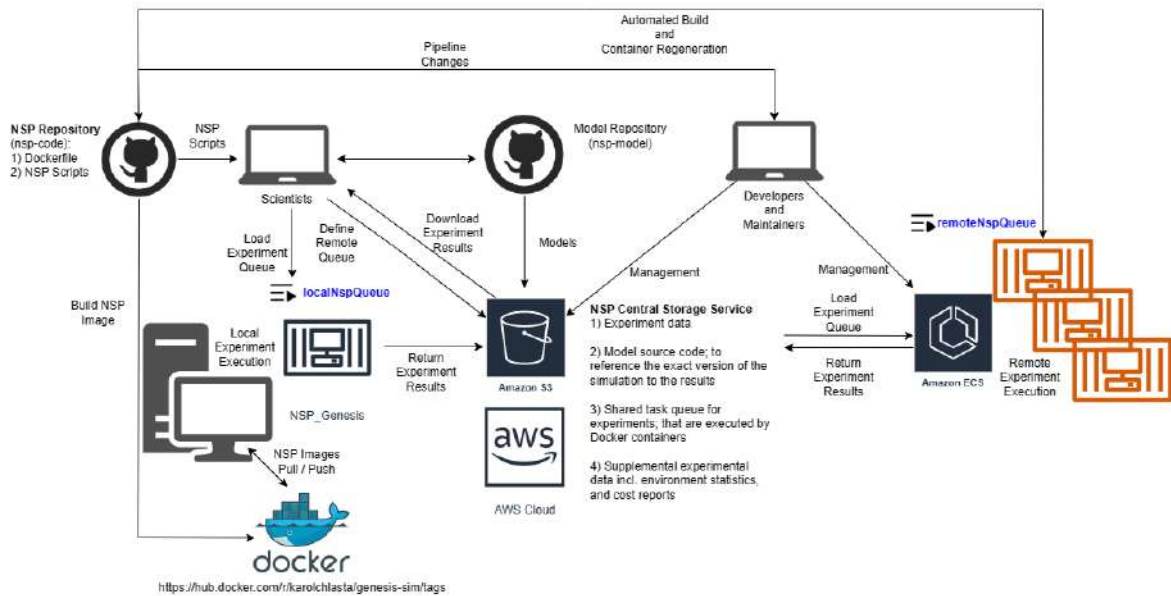


Fig. 5.5 Schematic diagram of Neural Simulation Pipeline showing how the pipeline is managed across the different components. Different types of users interact with the system via different interfaces.

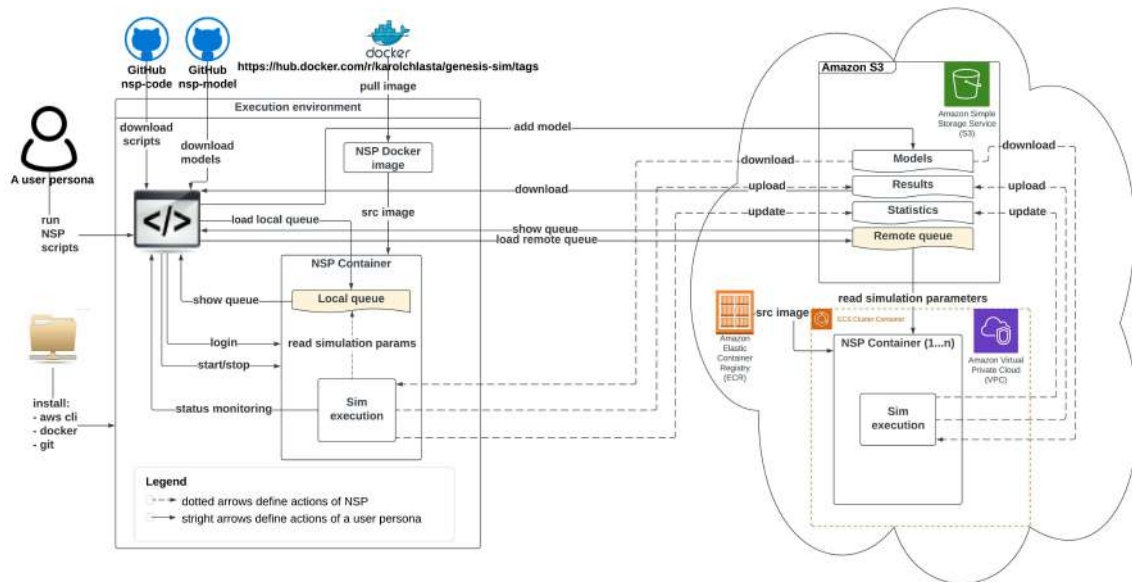


Fig. 5.6 Schematic diagram of Neural Simulation Pipeline showing a user workflow. This persona interact with the system via NSP scripts.

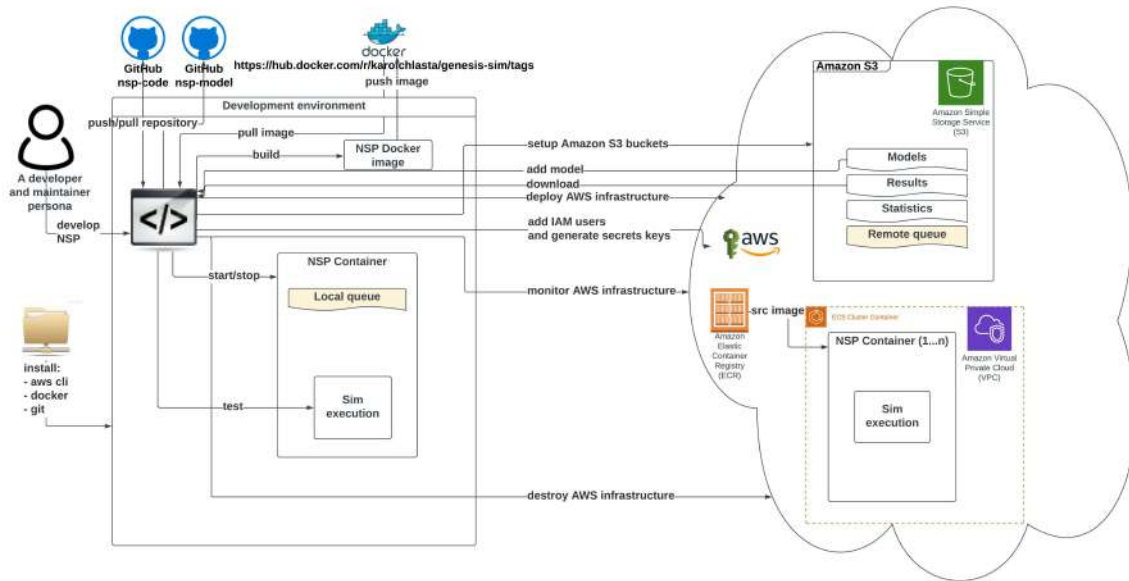


Fig. 5.7 Schematic diagram of Neural Simulation Pipeline showing a developer and maintainer workflow. This persona interact with the system via Git client (code), AWS CLI and AWS Management Console.

Figures 5.6 and 5.7 illustrate an architectural blueprint of Neural Simulations Pipeline. They explain how different elements of the pipeline run, and interact with the two categories of the users. The first, Figure 5.6 focuses on a user workflow, whereas Figure 5.7 highlights a developer and maintainer workflow; explaining what the key actions are, and if they are performed by the users or by the pipeline itself.

A user workflow in Figure 5.6 presents all the key user interactions with the NSP through solid lines and straight arrows. These key actions are: installing the software pre-requisites, downloading our NSP scripts, pulling our NSP Docker image from the DockerHub registry, starting a local container, defining a local or remote simulation queue, as well as monitoring the status of execution, and finally downloading the simulation results. After all the local simulations are finished it is a good practice is to stop the container to release system resources. The user workflow is supported by NSP User Scripts described individually in Subsection 5.4.2.

However, if a remote cloud-based execution is attempted, the NSP Docker image from the DockerHub is not needed. The pipeline provides the automated build and management of NSP image through a native Amazon ECR service, what guarantees the optimal performance and connectivity to other AWS services. All the repetitive actions related to data movement to/from a NSP container have been automated through NSP Container Scripts, described individually in Subsection 5.4.7; the Figure 5.6 shows these actions with dotted lines. The

figure also presents two simulations queues available in NSP (*localSimulationQueue.nsp* or *remoteSimulationQueue.nsp*, in yellow), with the local queue being active since the local container start, and the remote queue being checked for the simulation tasks in a definable interval.

Figure 5.7 shows key actions performed in a developer and maintainer workflow. The persona configures and operates the cloud-based execution environment. Our AWS NSP infrastructure is created on-demand via Terraform [30] using IaC approach. The Terraform configuration covers all the required services incl. the network components of VPC, Amazon Elastic Container Service cluster setup and the configuration of AWS CodePipeline, using AWS CodeBuild and AWS CodeDeploy. The AWS CodePipeline task is triggered automatically by a commit done to the 'main' branch of *nsp-code* repository. The AWS CodePipeline downloads the *nsp-code* repository from the of 'main' branch, and runs the AWS build task that executes Docker commands from the Dockerfile defined in the repository. As a result, the new NSP Docker image is created and pushed into the Amazon Elastic Container registry.

Figure 3.7 complements the architectural overview with a list of services needed to execute the simulations either (1) through the public cloud (AWS), or (2) using on-premise infrastructure. In both use cases, we adopt a central storage service (Amazon S3). That storage infrastructure holds:

- The simulation data (understood as both the input parameters, and the results of simulations).
- The model's source code, to reference the exact version of the simulation to the results.
- The task queue for simulations, that are executed by Docker containers.
- Supplemental experimental data incl. environment statistics, and cost reports.

Each type of execution requires provisioning a different set of services:

1. *Cloud execution* (upper part of Figure 3.7) - in this use case, we are using standard AWS services, allowing for an automated build of the Docker image (based on the *Dockerfile*, defining the compilation and installation steps of all the necessary libraries, and software components; file available in the main *nsp-code* repository). It provisions AWS CodePipeline, AWS CodeBuild, CodeDeploy, that produces the container image available in Amazon Elastic Container Registry. The container image is maintained through Amazon ECR service, with individual containers being managed through AWS Fargate engine. All the logs from simulation processing, as well as from the containers and the pipeline, are stored in AWS CloudWatch. After NSP is configured in the AWS cloud, the simulations can be run through AWS

containers using the remote simulation queue, that is defined as a text file at `s3://nsp-project/requests/remoteSimulationQueue.nsp`. This queue, when populated with a list of simulations, will be executed by AWS containers.

2. *On-premise execution* (lower part of Figure 3.7) - in this use case we execute simulations on own, or shared computer such as a laptop, workstation, or a computational cluster. As a result a local Docker client needs to be installed, and a public DockerHub service can be used to download our NSP image⁶ containing the latest, pre-configured version of GENESIS simulation engine.

There are three functional components of NSP: (1) simulation preparation, (2) simulation execution and (3) simulation post-processing. The preparation component manages the simulation's input data into a format suitable for the simulation, while the execution module performs the actual simulation execution using a selected engine. The current version of NSP also allows to select either the standard or parallel version of GENESIS simulation engine. The selection is performed via a parameter of `runSim.sh` script; with value `parallelMode = 1` indicating a parallel run. Finally, the post-processing module facilitates the analysis of output data, and generates the final results for a given simulation.

These components are built around two types of scripts. These are the *container scripts*, automating the simulation tasks within the application container, and the *user scripts*, responsible for the interaction with pipeline's end-user. Both types of scripts are summarised in Table 5.1 and described in Subsection 5.4.2 (user scripts) and Subsection 5.4.7 (container scripts). All the NSP scripts are installed automatically with our Docker image.

1. Developer creates new simulations scripts using GENESIS object model and programming syntax that are similar to these from object-oriented programming languages. New script file with the simulation code is created, a code change is committed to the source control repository.
2. Script `buildNspServerImage.sh` picks up the commits and triggers a new build.
3. Checkout of the code is initiated by the script and validation of code (if required) is done. The process of building the image is partially automated using the standard tools like AWS Pipeline.
4. Automated tests on the new image are run. A few sample experiments are run, so that the image is tested for expected results against; output files are validated (e.g. `experimentInfo.out`, `modelName.out` and `.dat` files)

⁶Official DockerHub NSP Image <https://hub.docker.com/r/karolchlasta/genesis-sim/tags>

Table 5.1 Full list of scripts in Neural Simulation Pipeline.

Component	User Scripts	Container Scripts
Simulation Preparation	buildNspImage.sh listModels.sh (.ps1) loadModels.sh (.ps1) pullNspImage.sh (.ps1) pushNspImage.sh runUnitTest.sh (.ps1) runUnitTestCheck.sh startNspContainer.sh (.ps1) getAWSCredentials.ps1	configAWSCLI.sh validatePositiveInteger.sh validateRealNumber.sh validateRange.sh
Simulation Execution	loginNspContainer.sh (.ps1) runSimLocally.sh (.ps1) runSimRemotely.sh (.ps1) runSampleSim.sh (.ps1) showNspQueue.sh (.ps1)	runSim.sh runSimLocally.sh runSimulationsManagerS3.sh showStat.sh showSystemInfo.sh calculatePeriod.sh writeDebug.sh writeOutput.sh
Simulation Post-processing	downloadSim.sh (ps1) listSim.sh (ps1) showNspContainerLogs.sh (.ps1) stopNspContainer.sh (.ps1) deleteNspImages.sh (.ps1)	downloadSim.sh downloadModel.sh listSim.sh saveStat.sh

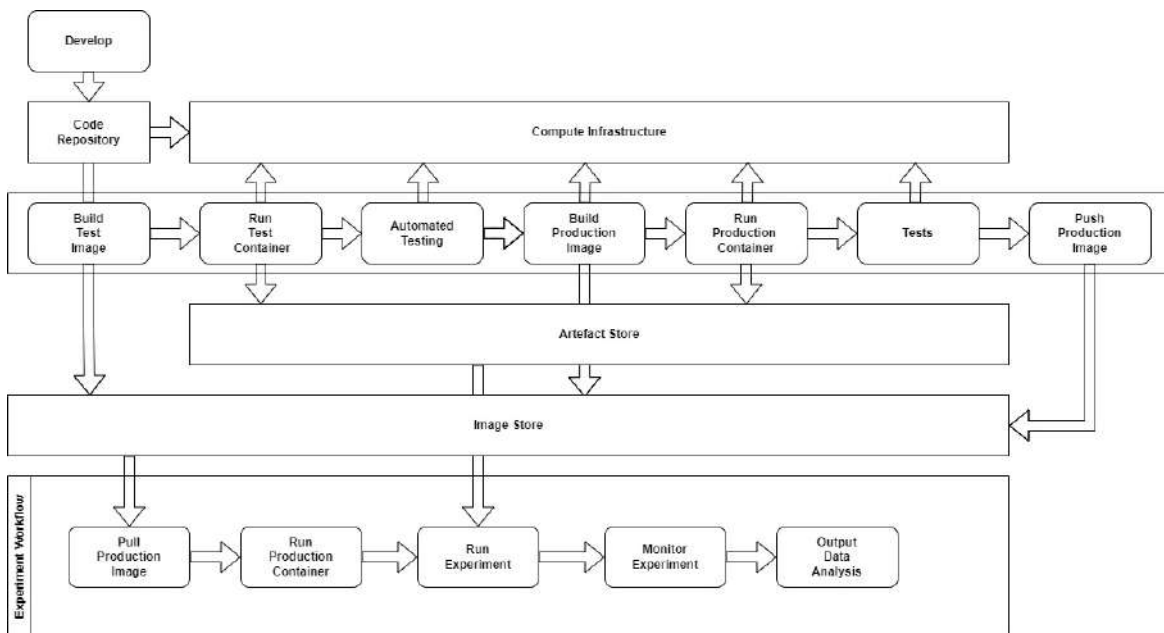


Fig. 5.8 Schematic, high-level view of Neural Simulations Pipeline.

5. Deployment to a Test environment is initiated within the Compute Infrastructure (using a Test container from the top of Fig. 5.8).
6. Full simulation tests of GENESIS programme are run against the model deployed to the test environment (e.g. target model, reduced simulation time for the neural system).
7. GENESIS programme is deployed to for a production run: on-premise or in the cloud (larger instance).
8. The NSP can be run for the experiments with or without containerisation.

The NSP facilitates an automated testing of the simulation code (models) through *runUnitTest.sh* and *runUnitTestCheck.sh*. These files contain sample tests. If a new model is developed, then the new test scripts might need to be created in an analogous way. Ideally the model will have full test coverage, what gives confidence that a given model is tested, and any bug is identified early in the development process. Applying this best practice is especially important for the long running brain simulations, whose bugs could often only be identified post-hoc e.g. after running for several days (or weeks) on expensive supercomputers. There are three scripts to facilitate parameter validation: *validateRange.sh*, *validatePositiveInteger.sh*, and *validateRealNumber.sh*. There are also other scripts supporting the simulation setup and execution. All the 34 Bash, and 16 PowerShell NSP scripts are listed by component in Table 5.1, as well as presented together with the cluster scripts in Appendix Table D.1.

The NSP scripts are also described individually in sections Neural Simulation Pipeline User Scripts (5.4.2) and Neural Simulation Pipeline Container Scripts (5.4.7) of this chapter.

As already mentioned, even the current version of NSP allows for a choice of simulation engines. The pipeline was tested with standard and parallel GENESIS simulation engines. The simulator is selected via the parameter of *runExp.sh* script with flag *parallelMode = 1* for a parallel run.

The pipeline also allows to define and reuse certain variables that are universal and independent from the simulation engine. We call them *NSP variables*. These variables should be added to the model's source code between the special character of "\$ \$" (e.g. "\$nspVariableName\$"). As a result the models' code can be more standardised, even across different simulation engines. Moreover, new possibilities could be created for boundary conditions investigation to improve brain simulation process, similarly to what was proposed in [83]. The current version of the pipeline recognises twelve NSP variables:

1. \$modelName\$
2. \$simSuffix\$
3. \$simDesc\$
4. \$simTimeStepInSec\$
5. \$simTime\$
6. \$columnDepth\$
7. \$synapticProbability\$
8. \$retX\$
9. \$retY\$
10. \$parallelMode\$
11. \$numNodes\$
12. \$modelInput\$

There are two types of statistics managed by the pipeline automatically through the *showSystemInfo.sh* NSP script generating the aggregated *simulationInfo.out* per simulation. These are:

- *Operating system level statistics* - describing the execution environment incl. process timings. These are generated using parameterised Linux commands of `date`, `uname`, `lshw`, `lscpu`, `lsblk`, `df`, `lspci` and `smem`. The script also uses `calculatePeriod.sh` subscript to calculate the exact time of simulation.
- *Simulation engine specific statistics* - they are triggered by the NSP through GENESIS `showstat` routine⁷.

The conceptual components in NSP are:

- **Testing Framework:** NSP image testing (regression testing); unit, integration, smoke testing (a group of new components combined to produce an output at a shorter simulation time) through dedicated scripts.
- **Source Control Management:** Author decided to use a Git repository to store the GENESIS programme code, hosted at GitHub, but any other source control management systems could be used. This repository will be synchronised with AWS S3 through scripts.
- **CI/CD tool:** NSP uses a partially automated AWS Pipeline for cloud deployment. Other tooling could be used (e.g. GitHub Actions), as long as it supports executing a shell command as part of the build.
- **Compute Infrastructure:** on-premise or in-cloud.

A best practice encouraged by the Neural Simulation Pipeline is Test Driven Development [9] (TDD). This practice requires writing tests before writing the GENESIS (or any other) simulation programme. Although such form of programming requires a shift in the mindset of a model developer, in author's view this additional effort is justified by the longer-term benefits brought into the project through improved standardisation and automation. To elaborate on that, the practice of preparing a bio-cybernetic model often requires considering the model functionality, and the output, before designing the final shape of a complex neural network. In such cases, NSP promotes so called "loose coupling", by allowing the model developer to start thinking outside a single simulation engine, through applying the set of unified NSP variables.

As presented in Fig. 5.8, there are a few steps in the pipeline and the Compute Infrastructure. In order to build a Neural Simulation Pipeline we conceptually also need the following items:

⁷GENESIS `showstat` routine <http://genesis-sim.org/GENESIS/Hyperdoc/Manual-25.html#showstat>

- *GENESIS programme tests*: these tests will call and validate all the elements of the simulation programme, so all script file that relate to a single GENESIS programme, and all the objects of a simulated cybernetic system. These tests build on the unit tests, and they will validate if a GENESIS programme code will run in context of the production like environment, prior to executing all the simulation steps on a target infrastructure.
- *Prototyping with a Sandbox*: the NSP tooling is not only a pipeline to deploy the simulation code, but it also allows to execute the simplified simulation (e.g. certain aspects of a cybernetic model), so that a model developer can run the simulation code from a local machine, as if the programme was deployed to a large supercomputer of a simulation cluster. This is useful if one needs to ensure that a model is properly tested prior to executing it on an expensive (cost-wise or time-wise expensive) computer infrastructure in order to test all aspects of the system, and to be able to see notifications about different experiment related statistics (e.g. about file storage or processes of the simulation). The NSP pipeline ensures that all the elements are tested using the predefined tests, and executed both on-premise, and in any remote cloud infrastructure.
- *Model branching support*: model developer might want to branch out a feature to work on its different version. NSP supports this by allowing to use different branches in the code repository. Each branch can be deployed to a different compute infrastructure.
- *Automated Tests*: the initial testing of the model code ensures that syntax and semantics of the model's code are of an acceptable quality. The next step is to ensure that all the remaining aspects of the system are tested prior to deployment of the model for a long term execution on a supercomputer or on a computational cluster. This process resembles load testing, and ensures that our model is capable of performing well in the target computational environment. These tests don't need to be executed every time, nevertheless they should run at least once prior to the deployment of model to the target computational environment.

To conclude, thanks to adopting a CD paradigm into the neuronal simulations, the quality of cybernetic models is expected to improve, as the model developers will receive feedback faster. As the NSP automates individual actions, these actions are performed faster and human error is eliminated. NSP therefore enables its users to benefit from the advantages of the modern development practices like TDD and elements of CI/CD processes.

The pipeline's source code is stored in *nsp-code* repository available publicly at GitHub⁸. This is the main application repository used for all the container builds, and it contains the *Dockerfile* describing the automated build process for GENESIS (in *nsp-server/Dockerfile*). The other repository used in the project is called *nsp-model*⁹. It stores the source code of all the RetNet models used in our simulations.

The pipeline's configuration is managed on different levels. The local Docker containers are configured through the *config.nsp* file, while the remote AWS containers are configured via Terraform configuration file (*modules\ecs-service\variables.tf*). The minimum required configuration includes the AWS access and secrets keys for authentication, as well as the basic metadata about the project incl. the scientist's name, surname, email. This information is automatically added to the simulation results. One of the useful NSP configuration parameters is a debug mode flag, enabled via *nsp_debug* parameter.

To summarise, author has built the NSP image for GENESIS simulator using the official Canonical Ubuntu bionic (version bionic-2022101) from DockerHub¹⁰. The automated build process installs: *csh*, *g++*, *libxt-dev*, *libxt6*, *libxtst6*, *libxtst-dev*, *libxmu-dev*, *mpich*, *gcc*, *bison*, *flex*, *libncurses5-dev*, *libxt-dev*. As a result, both GENESIS and its parallel version PGENESIS are compiled with all the dependencies, and our official, publicly available NSP image can be found in DockerHub. The image uses 424.83 MB and can be pulled from the DockerHub with the below command:

```
1 $ docker pull karolchlasta/genesis-sim:prod
```

Author welcomes new pushes of the updated NSP image with a 'test' tag to DockerHub¹¹, so that they can go through a review process, and can be made available to the other members of scientific community to facilitate their simulations.

Neural Simulation Pipeline User Scripts

Simulation Preparation Component

1. *buildNspImage.sh* - Bash script that clones the repository with the source code of cybernetic models (*nsp-model*¹²) and builds a Docker image of the simulator from a

⁸NSP Code Repository, containing a source code of the pipeline <https://github.com/KarolChlasta/nsp-code.git>

⁹NSP Model Repository, containing a source code of the simulations <https://github.com/KarolChlasta/nsp-model.git>

¹⁰Official DockerHub Linux Image https://hub.docker.com/_/ubuntu18.04

¹¹Official DockerHub NSP Image <https://hub.docker.com/r/karolchlasta/genesis-sim/tags>

¹²NSP Model Repository, containing a source code of the simulations <https://github.com/KarolChlasta/nsp-model.git>

DockerHub registry (*karolchlasta\genesis-sim*¹³) by compiling the simulation software with all the required libraries (from the 'main' branch) and building the container with the input tag name e.g. 'prod' (production) tag. The script generates locally usable container image with GENESIS and PGENESIS simulation engines, models and build logs. The build logs are named using the ID of the build process (unique) e.g. *docker_build17048.log*.

- *Inputs:*

- (a) Tag name for the container: 'prod', 'test'.
- (b) Branch name with source code for the image.
- (c) Login credentials to a remote Git repository with the source code of cybernetic model(s); only needed if the repository is not public. Currently: the password to the *nsp-model* repository.

- *Outputs:*

- (a) A ready to run Container image in the local repository of images.
- (b) Build logs in folder called 'docker_build_logs'.

2. *listModels.sh* - Bash script that shows the model's names loaded into NSP storage service in Amazon S3.

- *Inputs:* N/A.

- *Outputs:*

- (a) Model's names loaded into NSP storage service in Amazon S3.

3. *loadModels.sh* - Bash script that downloads the content of the repository with models (*nsp-model*¹⁴) and uploads them to the NSP storage service in Amazon S3.

- *Inputs:* N/A.

- *Outputs:*

- (a) Remote code repository with models copied to the NSP's storage service in Amazon S3.

4. *pullNspImage.sh* - Pulls the NSP remote image to the local Docker repository.

¹³Official Docker Image Distribution Registry <https://hub.docker.com/r/karolchlasta/genesis-sim>

¹⁴NSP Model Repository, containing a source code of the simulations <https://github.com/KarolChlasta/nsp-model.git>

- *Inputs:*
 - (a) Image tag name (default karolchlasta/genesis-sim:\$TAG_NAME).
 - *Outputs:*
 - (a) Image in the local Docker repository.
5. *pushNspImage.sh* - Bash script that pushes a tested image to the remote Docker registry (*karolchlasta\genesis-sim:input_tag_name*).
- *Inputs:*
 - (a) Tag name of the local image to be pushed to a remote Docker registry (repository of images).
 - *Outputs:*
 - (a) Container 'prod' image in the remote Docker registry.
 - (b) Build logs in the folder *docker_build_logs*.
6. *runUnitTest.sh* - Bash script that runs a few simple simulations to validate a new container image. The script is designed to automate a regression test, to check if the simulation engine works as expected, after the changes to the image definition have been applied.
- *Inputs:*
 - (a) The container's unique name; for the existing container, that we want to test on (default is the latest container created).
 - *Outputs:*
 - (a) Simulation results (all output files) uploaded to the Pipeline's storage service in Amazon S3.
 - (b) The file *historyOfSimulations.nsp* containing the execution history of all the batch simulation commands.
7. *runUnitTestCheck.sh* - Bash script that checks for the completeness of data generated by the simulation.
- *Inputs:*
 - (a) Name of simulation model.
 - (b) Name of simulation pattern.
 - (c) URI of simulation folder name with results stored in Amazon S3.

- *Outputs:*
 - (a) Simulation results uploaded to storage service in Amazon S3.
 - (b) Confirmation of the completeness of the simulations written to a standard output.
8. *startNspContainer.sh* - Bash script that starts the NSP container based on the image with a given tag. the name of the container will be *nsp_genesis*.
- *Inputs:*
 - (a) Tag name (default 'prod').
 - (b) Unique flag allows to create another container with an unique name *nsp_genesis_process_id*.
 - (c) Config file with NSP configuration.
 - *Outputs:*
 - (a) Running container.
9. *getAWSCredentials.ps1* - PowerShell script that gets an AWS session token and returns AWS credentials. Useful if the development machine runs on Windows.
- *Inputs:* N/A.
 - *Outputs:*
 - (a) AWS AccessKeyId.
 - (b) AWS SecretAccessKey.
 - (c) AWS SessionToken.

Simulation Execution Component

1. *loginNspContainer.sh* - Bash script that allows to login to the NSP container console based on the unique name or id.
- *Inputs:*
 - (a) The container's unique name or id; for the existing container, that we want to show console output (default the latest created container).
 - *Outputs:*
 - (a) Output of the Container's console.

2. *runSimLocally.sh* - Bash script for running NSP simulations in a batch mode on the local container. The script load input simulations file to the local queue of the input container. The script is designed to check if a simulation engine is running (by default every 3 minutes); if so it executes the next simulation command from the input *localQueue.nsp* file.
 - *Inputs:*
 - (a) The container's unique name; for the existing container, that we want to test on (default is the latest container created).
 - (b) The file *localQueue.nsp* with commands for a simulation engine.
 - *Outputs:*
 - (a) Simulation results uploaded to the Amazon S3.
 - (b) The file *historyOfSimulations.nsp* containing execution history of all the batch simulation commands uploaded into Amazon S3.
 - (c) The file *globalStatistics.nsp* containing summary of execution results of all the batch simulation commands uploaded into Amazon S3.
3. *runSimRemotely.sh* - Bash script for running NSP simulations in a batch mode in AWS cloud. The script load simulations file to S3 remote NSP file. Simulations will be taken by container for simulation. Script is running (by default every 3 minutes); if so it executes the next simulation command from the input *remoteSimulationQueue.nsp* file.
 - *Inputs:*
 - (a) The file *remoteSimulationQueue.nsp* with commands for a simulation engine.
 - *Outputs:*
 - (a) Simulation results uploaded to the Amazon Simple Storage Service (S3).
 - (b) The file *historyOfSimulations.nsp* containing execution history of all the batch simulation commands uploaded into Amazon S3.
 - (c) The file *globalStatistics.nsp* containing summary of execution results of all the batch simulation commands uploaded into Amazon S3.
4. *runSampleSim.sh* - Bash script for running a few sample NSP simulations directly by container simulator. Script is designed to help the user to quickly check the whether the configuration of NSP is correct.
 - *Inputs:*

- (a) The container's unique name; for the existing container, that we want to test on (default is the latest container created).
 - *Outputs:*
 - (a) Simulation results uploaded to the Amazon Simple Storage Service (S3).
 - (b) The file *historyOfSimulations.nsp* containing execution history of all the batch simulation commands uploaded into Amazon S3.
 - (c) The file *globalStatistics.nsp* containing summary of execution results of all the batch simulation commands uploaded into Amazon S3.
 - (d) Simulations results downloaded to the current folder.
5. *showNspQueue.sh* - Bash script for showing the content of remote or local container queue. The script allows to check how many simulations are waiting for execution.
- *Inputs:*
 - (a) parameter local indicating that we want to check how many simulations waiting in local queue for execution by the input container.
 - (b) parameter remotely indicating that we want to check how many simulations waiting in local queue for execution by the input container.
 - (c) The container's unique name; for the existing container, that we want to test on (default is the latest container created).
 - *Outputs:*
 - (a) content of container local queue or remote queue from Amazon S3.

Simulation Post-processing Component

1. *listSim.sh* - Bash script that shows the simulation's names loaded into NSP storage service in Amazon S3. Script requires AWS command line interface installed on the local computer and setup communication to AWS cloud via access and secrets keys: *aws_access_key_id*, *aws_secret_access_key*.
 - *Inputs:* N/A.
 - *Outputs:*
 - (a) Simulation's names loaded into NSP storage service in Amazon S3.
2. *showNspContainerLogs.sh* - Bash script that shows the console output of the running NSP container based on the unique name or id.

- *Inputs:*
 - (a) The container's unique name or id; for the existing container, that we want to show console output.
 - *Outputs:*
 - (a) Output of the Container's console.
3. *stopNspContainer.sh* - Bash script that stop and delete the NSP container based on the name or id.
- *Inputs:*
 - (a) The container's unique name or id; for the existing container, that we want to stop and delete (default latest created container).
 - *Outputs:*
 - (a) Deletion of the container.
4. *deleteNspImage.sh* - Bash script that delete NSP images based on the source repository *karolchlasta/genesis-sim*.
- *Inputs:*
 - (a) N/A.
 - *Outputs:*
 - (a) Deletion of the all NSP Docker Images.

5.4.3 Containerisation with Docker

Author believes that the problem of building, testing and deploying computer simulation models, as well as their different software dependencies could be resolved using a container platform like Docker [162]. Docker is the most popular container platform; as suggested in the recent IDC's white paper [40], it has already attracted "a significant amount of industry recognition", and it has the opportunity to "define the road map for all container platforms" as it runs on almost all CPU types and hardware platforms. It also has the potential to become a key component of "all the enterprise IT environments globally", due to its operating system neutrality. The same report mentions that "the container revolution is under way", with a forecast of 1.8 billion enterprise container instances deployed by 2021. Containerisation makes it possible to use provider-agnostic computing (IaC) in the way that the required

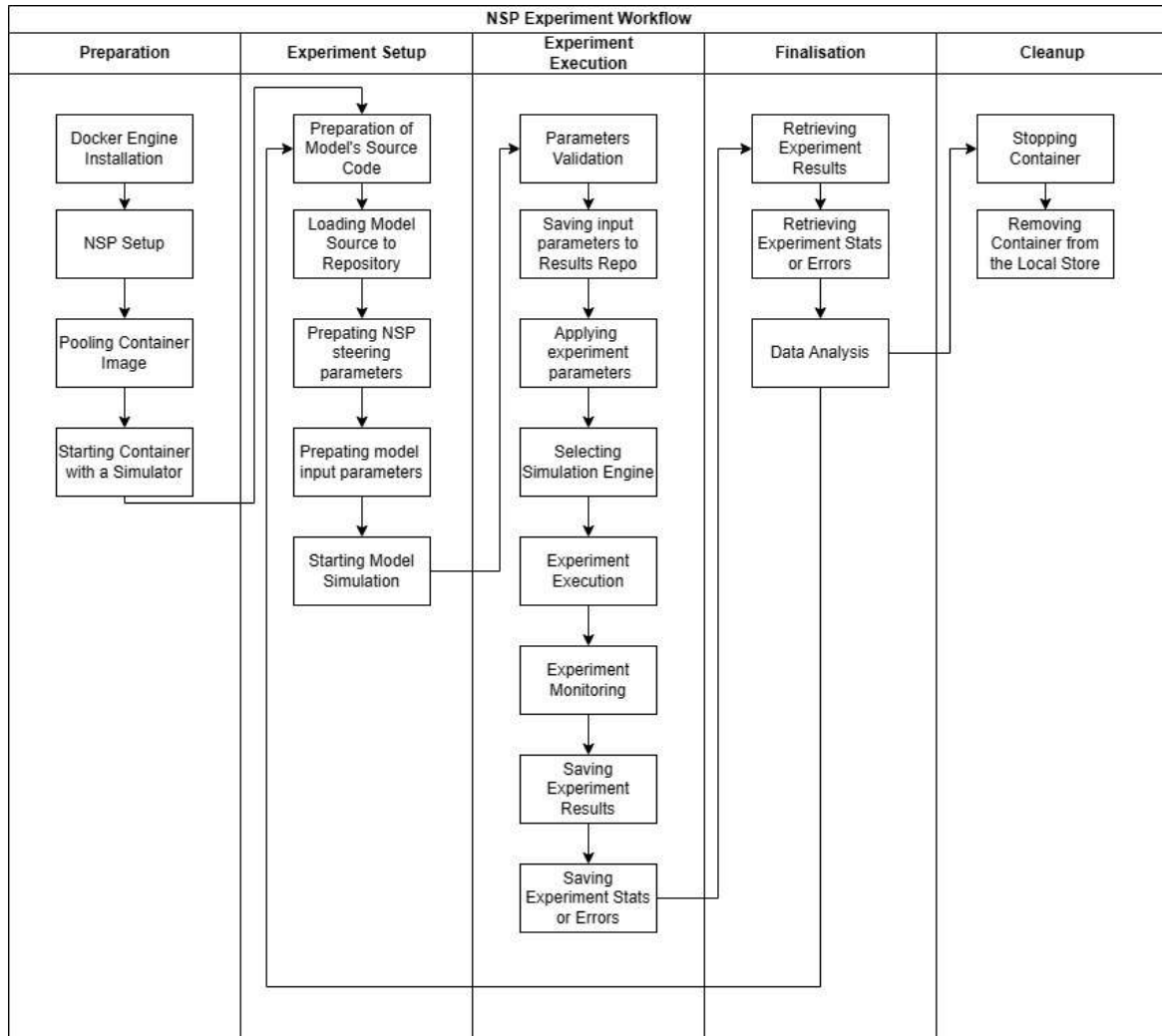


Fig. 5.9 Detailed experiment workflow in Neural Simulation Pipeline.

resources can be specified in a simple configuration file for multiple deployments to different hardware architectures [172].

As mentioned in the previous paragraph, Docker is growing in popularity and becoming a leading container platform globally. Nevertheless, in order to successfully use containers in a production environment, a *container orchestration* system is needed. Such a system “automates and simplifies provisioning, and deployment and management of containerised applications”. As per IBM Cloud Education (2021) [62], Kubernetes is the most popular container orchestration platform. The other popular container orchestration tools include Docker Swarm and Apache Mesos. Although the containers can be deployed and managed manually, most organisations automate the processes using pipelines [3].

There are several anticipated benefits of using a container platform. The top 7 of them have been gathered by Kumina¹⁵, a consultancy specialising in open source software since 2007, and container technology since 2014. In author’s view they present [38] a good summary of all the potential benefits, encouraging container adoption into any large scale research infrastructure for neural computations:

1. *Platform independence: build it once, run it anywhere.* Listed as a major benefit, because a container wraps up an application (e.g. neural simulation platform) with all its configuration files and dependencies, allowing to run these applications more easily in different environments (e.g. a development workstation, virtual servers, large computational cluster, in public or private clouds). This is also a key benefit for commercial organisations, who might seek an easy option to switch to another public cloud provider to avoid vendor locking for their infrastructure.
2. *Resource efficiency and density.* Each container does not require a separate operating system, as all containers share a single kernel. As a result, a containerised application is expected to use much less resources than a regular Virtual Machine (VM). A VM often requires several gigabytes (GB) in size, while a container usually measures from dozens to hundreds of megabytes (MB) [174]. Smaller size allows for running more containers than VMs using the same computing resources, what results in higher utilisation of hardware, and thus a reduction of data centre costs.
3. *Improved security through isolation with resource sharing.* Container platform provides isolation. In case an application crashes in one container, other containers with the same application will continue running. The isolation also decreases security risks e.g. in a scenario when the application is compromised or hit by a malware the negative

¹⁵Kumina’s website https://kumina.nl/why_docker_containers.

effects will not spread to the other running containers, and the affected container can easily be killed and/or recreated.

4. *Speed: start, create, replicate or destroy containers in seconds.* Containers are much smaller in size than VMs. They start in less than a second because they do not require an operating system to boot. The other operations of creating, replicating or destroying containers can also be completed within seconds. This speeds up the development and release processes. It might also improve customer experience e.g. through faster bug fixing in production.
5. *Immense and smooth scaling.* A major benefit of containers is that they offer the possibility of horizontal scaling, meaning that one can add more identical containers within a cluster to scale the application out. Container technology can provide so called “smart scaling”, the number of containers needed can be adjusted in real time, that reduce resource costs, or deliver the right size of a platform on-demand. Horizontal scaling has been used by all major public cloud vendors for years now [62].
6. *Operational simplicity.* As in the container world the host OS does not need any specific software or libraries to run its applications, it is easier to apply OS updates or security patches, and to perform vulnerability management across the estate.
7. *Improved developer productivity and development pipeline.* Applying containers eliminates environmental inconsistencies, what makes testing and debugging less complicated and less time-consuming. This is because there are fewer (or no) differences between development, test and production environments. The development pipeline is simplified, with the process of changing application infrastructure reduced to (1) the modification of the configuration file (often called a manifest file), (2) creating new containers and (3) destroying the old ones; a process which can be executed in seconds. Added version control makes it possible to roll-out or roll-back these environmental changes with "zero" downtime, and to share the stable and tested containers across different user groups faster.

Docker container platform is expected to be key to managing the ever-expanding “diversity of IT environments” resulting from multiple operating systems, different hypervisors, a combination of private and public clouds [40]. The same author suggests that “while Docker seems to be the solution for cloud-native, micro-services oriented applications, the platform is also well suited to many existing applications that could benefit from being modernised and prepared for possible refactoring over time”. In author’s view this statement applies also

to GENESIS [24] general purpose simulation platform (v2) that has been modernised to its version 3 at least since 2015 under the umbrella of Neurospaces project¹⁶.

This thesis uses Docker platform, notably its Personal subscription. The personal subscription is free for both education and research, as well as for businesses employing up to 250 employees, and whose revenue is less than \$10 million in per anum. The subscription (as of July 2022¹⁷) consists of:

1. Docker Desktop
2. Unlimited public repositories
3. Docker Engine + Kubernetes
4. 200 image pulls per 6 hours
5. Unlimited scoped tokens

Author selected this platform and this specific subscription, because it is both open source, and popular among in the community of users and developers; (which in author's view might result in its stability), and as such it allows to reduce the risks and complexity that using neural simulators often brings. In author's view this approach contributes to reducing the entry barriers to performing neural or (larger scale) brain simulations.

In spite of a wide enterprise adoption, there are significant problems with resource allocations [50] when using Docker containers on HPC platform, and running simulations using MPI communications with SLURM scheduler [238], a popular combination of tools used for large scale simulations. These problems are resolved using additional front-ends allocating the containers, or developing the alternative containerisation systems [10].

In this thesis author presents the *Neural Simulation Pipeline* (NSP), that is a simple alternative, that developed with Bash [185] and PowerShell [109], and does not require SLURM to execute simulations.

5.4.4 Docker Architecture

As summarised by Nickoloff and Kuenzlin [174], Docker platform uses the low-level kernel internals to run containers using the *Docker Engine*. The process is transparent for applications, as a container is a ring-fenced area of operating system with limits imposed on how much system resource it can use.

¹⁶Neurospaces Project <http://neurospaces.sourceforge.net/>.

¹⁷Docker Pricing & Subscriptions <https://www.docker.com/pricing/>.

The architecture of Docker containers relies on both *namespaces* and *control groups*. Their relationship for two sample containers can be seen in the Fig. 5.10. Historically the low-level Linux kernel constructs were available in Linux out of the box, but they were difficult to use. This gap was explored by the Docker Engine, whose authors have made using them easier. The engine was designed in a modular way, with two main interfaces: the Command Line Interface (CLI) and Application Programming Interface (API). The engine creates a layer of abstraction for all the required kernel internals, and creates a container that is designed for hosting specific applications and their dependencies [162]. This contributes to an increased level of automation within the application estate [3].

5.4.5 Docker Kernel Internals

Docker is multi-platform and works both under Windows and Linux. Docker containers have been around since 2013. Their beginning dates back to Linux kernel internals introduced to the kernel in 2009. Initially the use of Linux kernel internals required an in depth knowledge of the kernel programming, and hence it was not popular in the industry and among researchers. Docker containers on Linux are build on this stable foundation. The two main building blocks for Docker container platform are *namespaces* and *control groups*. Both of them are Linux kernel primitives. That's why it's widely assumed that the genesis of modern containers happened on Linux operating system [162].

Windows platform has had a few internal projects to implement containers, like *Draw-bridge* or *Server Silos*, but they have never been fully successful as admitted even by Microsoft itself [35]. This is the reason, why nowadays both Linux and Windows use Docker namespaces, which enforce isolation, and control groups that enforce their limits. This combined functionality allows to take an operating system and “carve it into multiple isolated virtual operating systems”. As such, the concept is similar to both hypervisor and virtual machine [162].

With virtual machine concept we take a single physical machine with all of its physical (hardware) resources, like CPUs and RAM, and define one or more virtual machines, each granted a slice of a virtual CPU, virtual memory, virtual networking, and virtual storage. In contrast, when using containers, we use namespaces take a single operating system with all of its resources, which tend to be the higher-level constructs like file systems and process trees or its users, and than carve all of them into multiple virtual operating systems called containers. As a result, each container is assigned its own virtual (also often called 'containerised') root file system, its own process tree, its own base eth0 interface, and its own root user. Although each container looks, and acts like an independent operating system (OS), they share a single kernel on the host machine. Containers introduce a layer of isolation, so that the processes

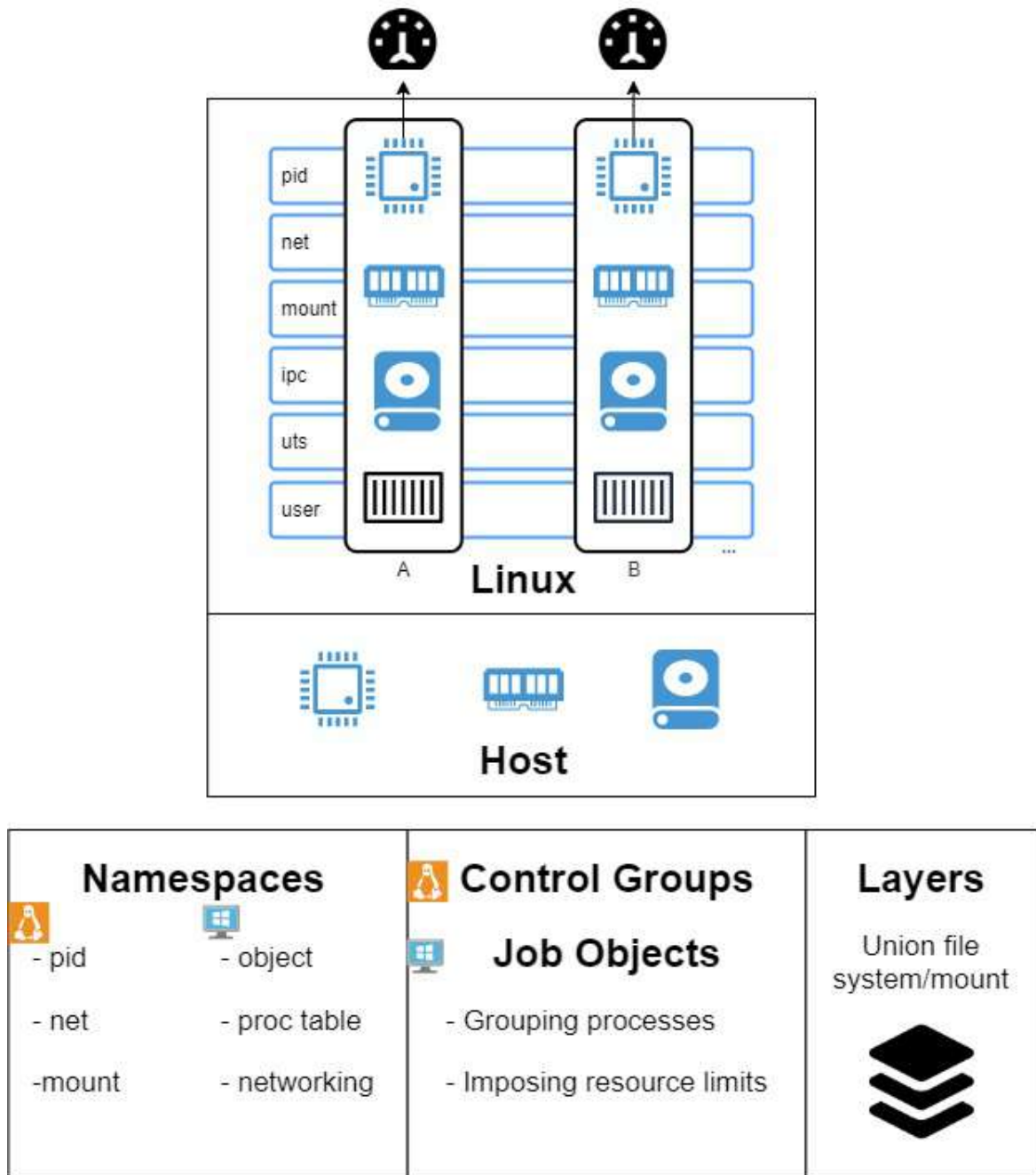


Fig. 5.10 Docker Architecture.

inside of a container don't know anything about any other processes in other containers. This architecture is summarised in Fig. 5.10, with two sample containers A and B.

A Docker container is an organised collection of namespaces, that has its own process ID (PID) table with PID 1, its own network namespace with an Eth0 interface, Internet Protocol (IP) address, and its own root file system [174]:

- The *PID namespace* gives each container its own isolated process tree, complete with its very own PID 1. In POSIX systems the init process owns PID 1 that is only responsible for starting and shutting down the operating system. This capability allows a process in one container to be completely unaware of any other processes.
- The *Net namespace* gives each container its own isolated network stack, by providing its network interface card (NIC), IPs and routing tables. Mount gives a container its own isolated root file system. Inter-process communication (IPC) lets processes in a single container access the same shared memory, and it stops all processes from outside of the container.
- The *User namespace* allows mapping accounts inside of a container into different users on the host. The typical example is to map the container's root user to a non-privileged user on the host.
- The *UNIX Timesharing System namespace* (UTS) gives every container its own host-name.

As a multi-tenant system is prone to a noisy neighbours problem, a solution to manage the consumption of system resources was needed. Linux has used control groups to achieve that since around 2009; the relevant kernel functionality was initially implemented to allow for resource auditing and limiting [162]. Windows developed this idea later, around 2015, with the introduction of Windows Job Objects approach [35]. The idea covering both operating systems is to group the processes and to impose limits on them. With namespaces and control groups implemented, both popular operating systems received the ability to host containers for production workloads.

The last fundamental component of a container platform is a *union file system*. Such a file system allows combining a few of read-only file systems or block devices with a write-able layer on top, and presenting them to the operating system in a unified view. These three elements: namespaces, control groups and union file system form the foundation for modern containers [174].

5.4.6 Docker Engine and NSP Scripts

Docker was created by a company called dotCloud. It originates from a Python tool called *DC*, that is an acronym for *dot* ('D'), and *cloud* ('C'). It was designed as a wrapper for *Linux Containers* (LXC) and aufs, with Aufs being a union file system, and LXC, a set of tools for interfacing with the container primitives in the Linux kernel.

The Docker Engine is at the core of Docker platform. It exposes Docker API to improve platform usability. It provides native orchestration with Swarm, on-premise secure registry, Universal Control Plane with its IT Ops user interface, and Role Based Access Control (RBAC) policies.

Docker platform uses *Docker client* to type in commands, and Docker daemon implementing the REST API, with *Containerd* being container supervisor handling execution of all the life-cycle operations, like start, stop, pause, and un-pause. The other component of the platform is the *OCI layer* that does the interfacing with the kernel. This is how the original, Linux version of the platform is designed. On Windows both the client and the daemon exist, but there is so called *compute services layer* instead of the containerd in the OCI layer. Both Linux and Windows versions of the Docker platform share exactly the same API. NSP uses several Bash scripts to interact with the Docker Engine.

5.4.7 Neural Simulation Pipeline Container Scripts

Simulation Preparation Component

1. *configAWSCLI.sh* - Bash script to configure AWS credentials. AWS credentials can normally be found in a user's folder (*.aws\credentials*), a single AWS configuration is needed per Region. This script might be useful if the development machine runs on Windows.
 - *Inputs*:
 - (a) AWS AccessKeyId.
 - (b) AWS SecretAccessKey.
 - (c) AWS SessionToken.
 - *Outputs*:
 - (a) Config files in user folder (*.aws\credentials*).
2. *validatePositiveInteger.sh* - Bash script that checks if the parameter is a positive integer.
 - *Inputs*:

- (a) Parameter to be checked.
 - *Outputs:*
 - (a) Text sent to the standard output.
 - (b) Return “0”, if the check passes; return greater than zero if the check fails.
3. *validateRealNumber.sh* - Bash script that checks if the parameter is a real number.
- *Inputs:*
 - (a) Parameter to be checked.
 - *Outputs:*
 - (a) Text sent to the standard output.
 - (b) Return “0”, if the check passes; return greater than zero if the check fails.
4. *validateRange.sh* - Bash script that checks if the parameter is inside the range.
- *Inputs:*
 - (a) Parameter to be checked.
 - (b) Begin of range (number).
 - (c) End of range (number).
 - *Outputs:*
 - (a) Text sent to the standard output.
 - (b) Return “0”, if the check passes; return greater than zero if the check fails.

Simulation Execution Component

1. *runSim.sh* - Bash script that runs a simulation. It specifically:
 - (a) Downloads the model (sources) from the storage service in Amazon S3.
 - (b) Reads and validates simulation parameters from the user, sets the default parameters.
 - (c) Parses the model source code, looking for the *NSP parameters*. The original model file is saved as an *.org* file.
 - (d) Builds the simulation name using the concatenated NSP parameters of date (%Y-%m-%d-%T), modelName, retX, retY, columnDepth, modelInput, simulation-Time, simulationsuffix (e.g. *2022-11-14-143707_RetNet40_5x8_75_0_1_HPIDocker*).

- (e) Runs the selected simulation engine (e.g. standard or parallel version).
 - (f) Prints the simulation execution status to the standard output.
 - (g) Saves environment statistics to the global statistics file.
 - (h) Saves the simulation files to the simulation folder in Amazon S3.
 - (i) Saves the simulation time (start and finish).
 - (j) Runs *showSystemInfo.sh*, and generates *simulationInfo.out* that contains all the parameters, start/stop time and environment statistics, as well as the screen output from the execution of the simulation, logs, and the errors (e.g. *ModelName.err* with errors).
- *Inputs:*
 - (a) Model name.
 - (b) Simulation suffix (to indicate execution environment).
 - (c) Simulation description.
 - (d) Simulation time step [s].
 - (e) Simulation time [s].
 - (f) LSM column depth (as described in Section 3.6).
 - (g) Synaptic probability (as described in Section 3.4).
 - (h) RetX (as described in Section 3.6).
 - (i) RetY (as described in Section 3.6).
 - (j) Parallel mode (“0” or “1” for parallel).
 - (k) Number of nodes (for parallel execution only).
 - (l) Model input pattern (input pattern, as described in Section 3.6).
 - *Outputs:*
 - (a) SimulationInfo.out
 - (b) ModelName.out
 - (c) ModelName.err
 - (d) All the relevant .dat files, as defined in the model.
2. *runSimulationsManagerS3.sh* - Bash script for running simulations in batch mode. The script checks (every 6 minutes) if a simulation engine runs; if not it executes the next simulation command from the *simulationsToRun.nsp* file on Amazon S3.
- *Inputs:*

- (a) File *simulationsToRun.nsp* with NSP steering commands for simulation engine(s).
 - *Outputs:*
 - (a) Simulation results uploaded to Amazon S3.
 - (b) File *historyOfSimulations.nsp* with the history of execution.
3. *showStat.sh* - Bash script that prints % of simulation execution based on a defined simulation step. Uses bash *grep* command on simulation's .dat files.
- *Inputs:* N/A.
 - *Outputs:*
 - (a) Text sent to the standard output.
4. *showSystemInfo.sh* - Bash script that prints the predefined OS statistics incl. information on hardware, kernel, memory, network, block devices, file systems, available disk space and active processes.
- *Inputs:* N/A.
 - *Outputs:*
 - (a) Text sent to the standard output.
5. *calculatePeriod.sh* - Bash script that prints the simulation time calculated from two timestamps.
- *Inputs:*
 - (a) start time of simulation.
 - (b) end time of simulation.
 - *Outputs:*
 - (a) simulation time sent to standard output.
6. *writeDebug.sh* - Bash script that prints text to the console output with *NSPDEBUG >* prompt depending on environment variable that is set in environment config file.
- *Inputs:*
 - (a) Text to be written in NSP output.
 - (b) *nsp_debug* environment variable.
 - *Outputs:*

- (a) Text sent to the standard output.
7. *writeOutput.sh* - Bash script that prints text to the console output with *NSP >* prompt.
- *Inputs:*
 - (a) Text to be written in NSP output.
 - (b) *nsp_debug* environment variable.
 - *Outputs:*
 - (a) Text sent to the standard output.

Simulation Post-processing Component

1. *downloadSim.sh* - Bash script to create a local copy (e.g. on the development workstation) of all the data for a given simulation, that is stored in NSP's cloud storage service (Amazon S3). Authorisation to the cloud resources happens via *configAWSCLI.sh*. All other NSP scripts use this script to download from the NSP's storage service.
 - *Inputs:*
 - (a) Amazon S3 URI (e.g. *s3://nsp-project/2022-10-16-182851_2neurons_test*).
 - *Outputs:*
 - (a) A folder with all the data for a given simulation.
2. *downloadModel.sh* - Bash script to download model code that is stored in NSP's cloud storage service (Amazon S3). Model folder is located under following URI path *s3://nsp-project/models/genesis/*
 - *Inputs:*
 - (a) AWS URI folder path to the models store (e.g. *s3://nsp-project/models/genesis/*)
 - (b) Model name (e.g. *2022-10-16-182851_2neurons_test*).
 - *Outputs:*
 - (a) A folder with all the data for a given model.
3. *listSim.sh* - Bash script to show which simulations are currently loaded to NSP's storage service. Simulation is defined as a folder, that contains all the simulation parameters, model(s) source code, outputs (all the data files), as well as logs and statistics.
 - *Inputs:* N/A.

- *Outputs:*
 - (a) Simulation names written to a standard output.
- 4. *saveStat.sh* - A Bash helper script that saves the global simulation statistics into the Amazon S3 object storage. The script takes the simulator's output (.out file), saves it as a single row and transfers the file to the global stats file in the AWS public cloud¹⁸.
- *Inputs:*
 - (a) Path to *globalStatistics.nsp* downloaded from Amazon S3.
 - (b) Output file of the simulator with the results of simulation.
 - (c) Input parameters of simulation.
- *Outputs:*
 - (a) *globalStatistics.nsp* updated and uploaded to Amazon S3.

5.5 Neural Simulation Pipeline Experimental Evaluation

In order to evaluate the NSP author has performed several full-experimental cycles and shown that the LSM models react differently to 3 different input patterns that are numbers (0, 1) and letter (A). The evaluation involved performing a total of 54 simulations on the RetNet models on which this PhD thesis reports. Author used NSP to measure the model execution time (CPU Time), memory consumption as well as the number of spikes in each simulation run. The exact results of these simulations are presented in Table 4.4 and Table 4.5. All the results and accompanying statistics have been gathered through running NSP scripts throughout the year 2022.

These aggregated, average results are presented in Figure 5.11, and Figure 5.12. They vary significantly, depending on the model complexity (number of HH neurons) and the execution environment, hence they were averaged per model. As a result, the figures present how each execution environment performs against that average. The simulation execution time (as measured by CPU time in seconds) varies from 2 minutes RetNet(8x5,1,25) to 28 hours RetNet(8x5,1,300) for the on-premise execution at HPI; and from 1 second for RetNet(8x5,1,25) to 24 hours for RetNet(8x5,1,300), when run as containerised at HPI; and from 4 seconds till 11 hours for the containerised AWS execution.

The AWS execution is over two times faster than the two alternatives at HPI. This pattern is additionally confirmed by the speed of Docker image builds. For the five Docker image

¹⁸NSP's object storage at Amazon S3: s3://nsp-project/simulations/globalStatistics.nsp

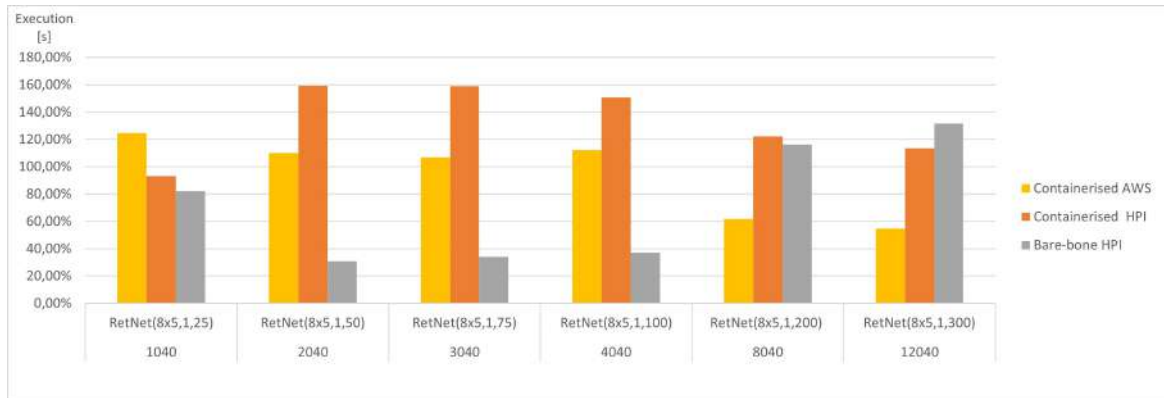


Fig. 5.11 Average CPU time for each RetNet model (ranging in complexity from 1040 to 12040 Hodgkin–Huxley neurons) viewing 3 input patterns (0, A, 1), expressed as a percentage of average CPU time across all execution environments for each RetNet model. Based on Table 4.4 and Table 4.5.

builds, the average *NSP_Genesis* container build time was only 5.3 minutes at AWS, whereas the same build at HPI took 12.40 minutes. Again, we notice a two-fold difference, which is surprising, assuming a “similar” simulation setting.

The memory utilisation (as measured by RAM consumed) varies significantly from 19 MB for RetNet(8x5,1,25) to 16 GB for RetNet(8x5,1,300) execution on-premise at HPI; from 26 MB for RetNet(8x5,1,25) to 4.6 GB for RetNet(8x5,1,300) executing through a container at HPI, and from 68 MB to 17 GB for the containerised AWS execution. In the case of memory consumption, author notices that the on-premise (direct) HPI execution is similar to the containerised execution at AWS. Surprisingly, that consumption for the containerised HPI execution is four times smaller, than in the other execution environments. On the other hand, the memory utilisation for the smaller models (so with a neural column depth of 50, 75 and 100), that executed on-premise, without the container at HPI is four times smaller if compared with their containerised execution at HPI.

Author compared the standard and containerised simulation setup on the same underlying hardware. The results measured on the HPI on-premise infrastructure do not indicate any major negative impacts of containerisation on the overall simulation performance. The average time (CPU time) needed to complete the containerised simulations of our RetNet models is 96.15% of the average simulation time needed to complete the same simulation on the virtual machine. Interestingly, the opposite was measured for memory consumption, the containerised simulation consumed 292% of the memory needed for a standard execution. The performance overhead of containerised execution is invisible, so running computationally intensive neural simulations seems even more appealing, especially assuming the scalability and affordability of public cloud execution environments [92].

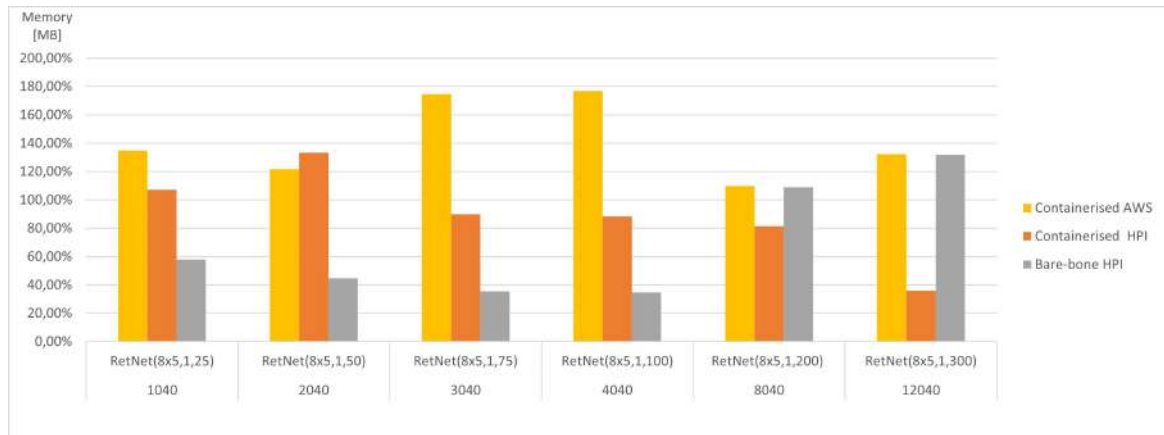


Fig. 5.12 Average memory for each RetNet model (ranging in complexity from 1040 to 12040 Hodgkin–Huxley neurons) viewing 3 input patterns (0, A, 1), expressed as a percentage of average Memory utilisation across all execution environments for each RetNet model. Based on Table 4.4 and Table 4.5.

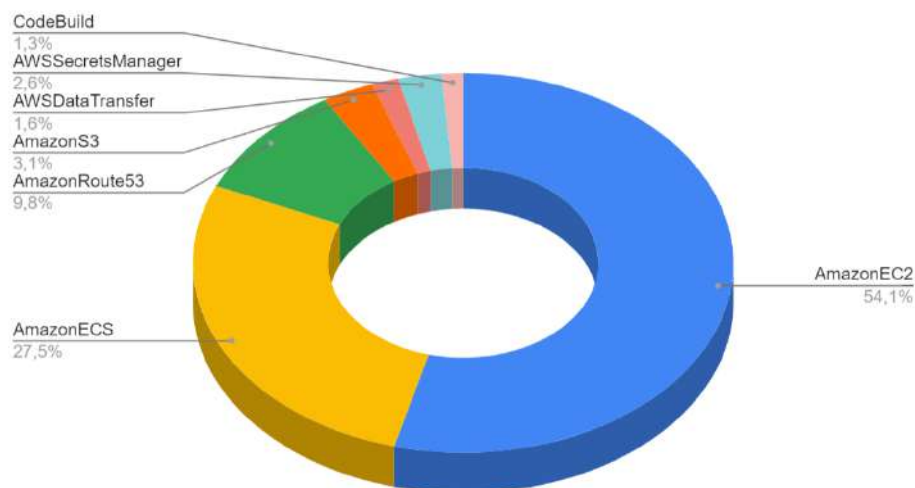


Fig. 5.13 Cost structure for 18 experiments run using Neural Simulation Pipeline on AWS Public Cloud infrastructure.

The execution of simulations with NSP in AWS public cloud environment allowed the author to investigate the cost per simulation, as well as the overall cost structure for the RetNet models. The overall cost structure is presented in Figure 5.13. We measured that 81.6% of the total cost is spent on AWS compute services (AWS ECS and Amazon EC2 spot instances). The rest of the cost is attributed to non-computational services: 3.1% on data storage (Amazon S3), 9.8% on Domain Name System (AmazonRoute53), 1.6% on data transfer, secure connection to GitHub 2.6% (AWS Secrets Manager), 1.3% on automation (CodeBuild).

Author has also calculated the real cost of each simulation. Simulating a single second of 1040 HH neurons using RetNet(8x5,1,25) costs on average USD 0.02, while the most expensive RetNet(8x5,1,300) built with 12040 HH neurons costs USD 4 to execute. A detailed cost per simulation is provided in Table 4.5.

5.6 Neural Simulation Pipeline Limitations

There are also a few other limitations in the current version of the Neural Simulation Pipeline. Firstly, the current version, first official NSP Docker image with the latest version of GENESIS simulation engines is relatively large. It requires 1.17 GB in the local repository, and 424.83 MB in the remote registry (that is after compression, at DockerHub¹⁹). Author believes that the image could be optimised by removing some non-critical operating system tools and utilities.

Secondly, the current Neural Simulation Pipeline supports GENESIS and PGENESIS [24] only. Author would like to create a version of the pipeline for each major [215] simulation engine like BRIAN [87], NEST [82] and NEURON [103]/CoreNEURON [137]; as well as other (e.g. functional) simulators like Nengo [17]. This will require a preparation of new *Dockerfile* in the *nsp-code* repository.

Thirdly, at present all the NSP containers are configured to read the file with simulation tasks from the Amazon S3 bucket at different moments in time. Nevertheless, a few containers could theoretically fetch the same simulation, if they hit the file exactly at the same moment in time. In the future, author wants to implement a proper semaphore mechanism for allowing, or disallowing access to the simulation task. This problem could potentially also be resolved using an Amazon SQS, a standard first-in-first-out (FIFO) queue service. Moreover, the NSP proof-of-concept was tested with only the three containers reading the remote queue, and

¹⁹Official Docker Image Distribution Registry <https://hub.docker.com/r/karolchlasta/genesis-sim>

executing the simulations in parallel. More containers could be evaluated to report a detailed performance of the solution.

Finally, author thinks that an important limitation of the NSP is that it does not provide a service for multiple research teams at the same time, and does not enable interdisciplinary work between different profiles of researchers. This additional functionality would likely require a web-interface for simulation management, as well as the security model based on a defined set of access rules for each role e.g. different for model developers and neuroscientists.

Chapter 6

Summary and Conclusion

6.1 Summary

Neural simulation is a computational approach that involves building and running computer models of the structure and function of the brain or parts of the brain. It can be used to study the brain and how it works, as well as to explore and test hypotheses about brain function in health and disease. Using neural simulation can be useful in studying and understanding the complexity of certain CNS disorders, as it allows researchers to investigate and analyse the brain's structure and function in a controlled and precise manner [66]. This can help to identify potential targets for therapeutic intervention and to test the effects of different treatments or interventions on brain function. The large-scale simulations of biologically realistic neural networks require large and expensive computational resources. Liquid State Machines [147] are important in brain modelling and increasingly important in different engineering applications [223].

The first goal of the study was to build a cybernetic model of the visual system using a spiking neural network that would be simple enough to execute on SBCs, allow for easy understanding of its operations, and at the same time it could illustrate the most important functions of the visual cortex.

This thesis analyses the body of knowledge related to modelling the networks of spiking neurons, and the visual system. It described the fundamental structures and mechanisms involved in (1) the interaction of the different elements of the visual system and the visual cortex in the brain, (2) discussed the computational models of neurons used in the process and simulation frameworks, (3) presented its possible implementation using Liquid State Machines, (4) performed simulations of neural models illustrating some aspects of the visual system modelled as Liquid State Machine.

The work is interdisciplinary, inspired and based on the analysis of numerous publications related to neuroinformatics, computational neuroscience and biopsychology (e.g. in the context of brain anatomy).

As the main contribution the author presents the *Neural Simulation Pipeline* (NSP). This novel experimental setup allowed running neural simulation in both the cloud environment and on-premise. In the background of the main work related to this PhD thesis, the author was experimenting with a low-power computational cluster called *Neural Simulations Cluster* (NSC) built with Raspberry Pi and ROCKPro64 boards to support the simulations in author's home environment. NSC allows developing and running parallel neural simulations at a fraction of the time and cost of running them on a dedicated supercomputer, or even a high-end workstation computer, while allowing for a more realistic development experience in the same way it would be happening on large scale computational cluster, or a dedicated supercomputer.

The NSP, a simple scientific workflow management system, based on a set of 18 Bash scripts, that manages experiments and facilitates defining and executing them across different simulation engines in a unified way. The authors managed to validate NSP by running it in three different types of run-time environments (1) using containers in the AWS cloud, and on-premise (2) on an HPI infrastructure and (3) directly on the operating system without containerisation. This simple scientific workflow system has also successfully managed the experiment queue, unified key experimental variables, collected data and experimental statistics, as well as provided basic validation of experimental parameters, monitored experiment execution, supported simulation code testing, and checked for the completeness of the experiment results.

In order to evaluate the NSP author performed several full-experimental cycles and shown that our LSM models react differently to 3 different input patterns that are numbers ("0", "1") and letter ("A"). NSP was used to measure the model execution time (CPU time), memory consumption as well as the number of spikes in each simulation run for the total of 54 experiments on the RetNet models reported in Chapter 4 of this thesis.

6.2 Future Directions

The current NSP requires a basic knowledge of the computer operating systems, ability to run Bash or PowerShell scripts (and working knowledge of some AWS cloud services). In future, the author would like to create a web application providing a simulation service using NSP containers without the need for running any scripts. This would allow to expose the NSP as a Scientific Workflow Management System to a wider community, gather feedback,

and potentially also allow to perform a more extensive testing with (other than AWS) public cloud services providers.

Author believes that NSP could lead to the enhanced planning and forecasting of costs for the large-scale simulations across different public clouds. In the current version, the author has only used the AWS Cost Reports, as a source of the cost information. Author envisions that a trial run in a public cloud could help the computational neuroscience researchers with their cost estimation. An automated trial run of a smaller model could also be a good proxy for a full scale execution, and it would allow both easier and more accurate budgeting, apart from just providing the researchers with simulation management and execution capabilities.

Author would like to create a version of the pipeline covering each major [215] simulation engine like BRIAN [87], NEST [82] and NEURON [103]/CoreNEURON [137]; as well as other (e.g. functional) simulators like Nengo [17].

Looking at these plans, author recognises that some simulators may suit better for running in containers than the others [50]. Considering that, author thinks that the next simulator to incorporate into the NSP is NEURON [103]/CoreNEURON [137]. It is the most popular software for brain network simulations, if counting the number of entries in ModelDB [104]. Moreover, NEURON's architecture and installation¹ resembles that of GENESIS, with the simulation setup requiring additional MPI libraries for parallel simulation.

The following one would be NEST², slightly less popular, but capable of running thread-parallel simulations “out-of-the-box” on multiprocessor computers with OpenMP [47]. For NEST, the application of NSP could benefit scientists, who would want to execute distributed simulations using MPI libraries.

Finally, although the BRIAN software has a monolithic architecture, it does not use external modules or libraries, and as such it also does not use MPI parallelisation. The benefit of using NSP could be in enabling this software to run simulations in parallel, on multiple nodes through the mechanism of NSP queues.

The other area of future development for NSP is the adoption of the ModelDB [164], rather than the approach involving the *nsp-model* GitHub repository. As a result, the process of inserting a new model into the NSP could happen directly from the ModelDB database in a standard way [104].

Apart from the scientific workflow, this PhD thesis described the structure and main elements of the visual system from a biocybernetic perspective. In the subsequent research author would like to simulate much larger models e.g. those built with millions of spiking

¹NEURON Documentation https://nrn.readthedocs.io/en/8.2.2/install/install_instructions.html

²NEST Guide for Parallel Computing https://nest-simulator.readthedocs.io/en/stable/guides/parallel_computing.html

HH neurons, as well as to present a more in-depth evaluation of the NSP using services provided by different public cloud services providers.

6.3 Practical Significance

As already mentioned, the large-scale simulations of biologically realistic neural networks require expensive computational resources [155, 66]. They create challenges with storing the massive amounts of their data [66], as well as with developing, distributing, and maintaining their cybernetic model codebase [49]. Both their configuration, and model deployment process might present a significant barrier for many researchers tackling biocybernetic modelling. This is due to both methodological and IT challenges [66], that include maintaining software dependencies and executing computations on different hardware infrastructures. The task is not trivial from a technical perspective, even when using such a well established simulation engine as GENESIS [44].

The practical significance of NSP is in reducing entry barriers to numerical systems modelling using the mid to large-scale simulations; with applications to both brain networks (GENESIS), and brain bio-mechanics (Kinetikit). The framework could also be used to improve experiment budgeting. NSP hides the complicated technical aspects of installing and configuring simulation engines on different platforms, enabling the same model to be easily run on-premise on different types of processors, and in-cloud using a predefined set of service parameters. The system is intended to popularise the use of computer simulators in brain research through its functionality summarised below.

NSP manages simulations and allows them to be saved and defined for different simulation engines in a unified way. The framework provides both *local and remote queues* for executing simulations. These queues can be executed regardless of the hardware platform, through Docker containers running in the cloud, or on-premise.

The NSP also enables *faster and easier* analysis of experimental data. This is because of four key reasons listed below:

1. All the simulation results are stored centrally using a single storage service.
2. They can be managed using a set of standard NSP scripts on both Linux and Windows machines.
3. Cybernetic models can be equipped with the reusable *NSP variables*, independently of a simulation programming language used.

4. All the experimental data get partially pre-processed, by aggregating the results together with the statistics on the execution environment (e.g. simulation run-times, detailed information about CPUs, memory and operating system processes, as well as simulation engine specific statistics).

To summarise, the practical significance of NSP would be in reducing entry barriers to numerical systems modelling and large-scale simulations through a Docker-based pipeline, that could be executed across multiple compute infrastructures. The practical application of the novel LSM-based RetNet system could be in pattern classification tasks for the industrial or medical setting.

6.4 Key Insights

This PhD thesis explores the simulation setup in computational neuroscience. Author uses GENESIS, a general purpose simulation engine for sub-cellular components and biochemical reactions, realistic neuron models, large neural networks, and system-level models. GENESIS supports developing and running computer simulations, but leaves a gap for setting up larger and more complex models of today. The field of realistic models of brain networks has outgrown the simplicity of earliest models. The challenges include managing the complexity of software dependencies and various models, setting up model parameter values, storing the input parameters alongside the results, and providing execution statistics.

Moreover, in the High Performance Computing context, public cloud resources are becoming an alternative to the expensive on-premises clusters. Author presents the Neural Simulation Pipeline, that facilitates the large scale computer simulations and their deployment to multiple computing infrastructures using the infrastructure as code (IaC) containerisation approach. He demonstrates the effectiveness of NSP in a pattern recognition task programmed with GENESIS, through a custom-built visual system, called RetNet(8x5,1) that uses biologically plausible Hodgkin–Huxley spiking neurons. The pipeline is evaluated by performing 54 simulations executed on-premise, and through the Amazon Web Services public cloud. The PhD thesis reports on the non-containerised and containerised execution with Docker, as well as presents the cost per simulation in AWS. In author's view the results show that the Neural Simulation Pipeline can reduce entry barriers to neural simulations, making them more practical and cost effective.

The results confirm no significant overhead of containerisation on CPU time for the RetNet model. The containerised execution was actually faster, taking only 96.15% of the average simulation time needed to complete the same simulation on the virtual machine. Interestingly, the opposite was measured for the memory consumption, the containerised

simulation consumed only 292% of the memory needed for a standard execution. The performance overhead of containerised execution is negligible, that is promising for its future applications in neural simulations.

Author also measured that the simulation of his own biological visual system that was built of a significant number of 12040 HH neurons executed for 11.62 hours in AWS public cloud environment, and costed USD 4 only. The other finding was that only 81.6% of the total cost spent on AWS compute services was actually spent on real computations (AWS ECS and Amazon EC2 spot instances).

Finally, the novel LSM system presented in this PhD thesis (called RetNet), whose computational complexity was estimated to grow in polynomial time $O(0.000679468 * n^2 - 1.09334 * x + 716.782)$, achieves both the accuracy and F1 Score of 81% with the readout function based on Light Gradient Boosting Machine algorithm.

6.5 Conclusion

In his thesis, the author claimed that numerical simulations must integrate a robust model development methodology, with adequate testing and simulation steering workflows to increase scientific throughput, and improve utilisation of current and next-generation computational infrastructure, available both on-premise and in-cloud. Author proposed to transform the end-to-end computational experiment workflow from one that is non-universal and manual to one that is standardised and automated through the novel *Neural the Simulation Pipeline*.

The pipeline, a simple scientific workflow management system, based on Docker and a set of 50 scripts (34 written in Bash and 16 in PowerShell) manages simulations and facilitates defining and executing them across different simulation engines and execution environments in a unified way. The author validated the NSP by running it in three different types of run-time environments: (1) using containers in the AWS cloud, (2) on-premise using the HPI infrastructure and (3) directly on the operating system without containerisation. This simple scientific workflow system has also successfully managed the simulation queue, unified key experimental variables, collected data and experimental statistics, as well as provided basic validation of experimental parameters, monitored simulation execution, supported simulation code testing, and checked for the completeness of the simulation results.

The practical significance of NSP is in reducing entry barriers to the numerical systems modelling and large-scale simulation execution through a Docker-based pipeline across multiple computing infrastructures. The NSP provides a set of tools for automating the build of GENESIS and PGENESIS from their source code to container images. The simulation engines are bundled with all the necessary software libraries, what allows for flexible testing

and deployment of simulations (understood as different cybernetic models) according to the IaC principle.

NSP tools also facilitate the analysis of experimental data. All simulation results are stored centrally, and available in a single, on-line storage. The experimental data gets partially preprocessed, that facilitates further analysis by aggregating the results and enriching them with additional information on the details of the execution environment, and run-time statistics (e.g. detailed run-times, information on microprocessors, memory and operating system processes).

Author evaluated the NSP using own Liquid State Machines RetNet models of up to 12040 HH neurons. The thesis explains how the containerised Docker-based pipeline designed by the author allows the simulations to be developed, tested and simulated in either an on-premise environment or in the public cloud environment. Finally, the thesis describes the application of the novel idea for simulation management, that simplifies model development and simulation across multiple execution environments. Author integrates this idea into the Neural Simulation Pipeline, and applies it into his own simulations.

To summarise, the significance of NSP is in reducing entry barriers to numerical systems modelling and large-scale simulations, with application to both brain networks (GENESIS), and brain bio-mechanics (Kinetikit) simulations. The framework could also be used to improve experiment budgeting. NSP hides the complicated technical aspects of installing a simulation engine on different platforms, enabling the same model to be easily run on different types of processors and in cloud computing with predefined service parameters. Author hopes that the NSP presented in this PhD thesis will popularise the use of neural simulators, and advance scientific workflows in brain research.

References

- [1] Abraham, A., Pedregosa, F., Eickenberg, M., Gervais, P., Mueller, A., Kossaifi, J., Gramfort, A., Thirion, B., and Varoquaux, G. (2014). Machine learning for neuroimaging with scikit-learn. *Frontiers in neuroinformatics*, page 14.
- [2] Ahern, S., Shoshani, A., Ma, K.-L., Choudhary, A., Critchlow, T., Klasky, S., Pascucci, V., Ahrens, J., Bethel, E. W., Childs, H., et al. (2011). Scientific discovery at the exascale. report from the doe ascr 2011 workshop on exascale data management. *Analysis, and Visualization*, 2(3).
- [3] Al Jawarneh, I. M., Bellavista, P., Bosi, F., Foschini, L., Martuscelli, G., Montanari, R., and Palopoli, A. (2019). Container orchestration engines: A thorough functional and performance comparison. In *ICC 2019-2019 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE.
- [4] Alivisatos, A. P., Chun, M., Church, G. M., Greenspan, R. J., Roukes, M. L., and Yuste, R. (2012). The brain activity map project and the challenge of functional connectomics. *Neuron*, 74(6):970–974.
- [5] Aloisio, G. and Fiore, S. (2009). Towards exascale distributed data management. *The International Journal of High Performance Computing Applications*, 23(4):398–400.
- [6] Amorim Da Costa, N. M. M. and Martin, K. (2010). Whose cortical column would that be? *Frontiers in neuroanatomy*, 4:16.
- [7] Amunts, K., Knoll, A. C., Lippert, T., Pennartz, C. M., Ryvlin, P., Destexhe, A., Jirsa, V. K., D’Angelo, E., and Bjaalie, J. G. (2019). The human brain project—synergy between neuroscience, computing, informatics, and brain-inspired technologies. *PLoS biology*, 17(7):e3000344.
- [8] Aradi, I. and Érdi, P. (2006). Computational neuropharmacology: dynamical approaches in drug discovery. *Trends in pharmacological sciences*, 27(5):240–243.
- [9] Astels, D. (2003). *Test driven development: A practical guide*. Prentice Hall Professional Technical Reference.
- [10] Azab, A. (2017). Enabling docker containers for high-performance and many-task computing. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pages 279–285. IEEE.
- [11] Bachatene, L., Bharmauria, V., and Molotchnikoff, S. (2012). Adaptation and neuronal network in visual cortex. *Visual Cortex-Current Status and Perspectives*.

- [12] Barak, O., Rigotti, M., and Fusi, S. (2013). The sparseness of mixed selectivity neurons controls the generalization–discrimination trade-off. *Journal of Neuroscience*, 33(9):3844–3856.
- [13] Basford, P. J., Johnston, S. J., Perkins, C. S., Garnock-Jones, T., Tso, F. P., Pezaros, D., Mullins, R. D., Yoneki, E., Singer, J., and Cox, S. J. (2020). Performance analysis of single board computer clusters. *Future Generation Computer Systems*, 102:278–291.
- [14] Becchetti, A., Gullo, F., Bruno, G., Dossi, E., Lecchi, M., and Wanke, E. (2012). Exact distinction of excitatory and inhibitory neurons in neural networks: a study with gfp-gad67 neurons optically and electrophysiologically recognized on multielectrode arrays. *Frontiers in neural circuits*, 6:63.
- [15] Beeman, D. (2005). Genesis modeling tutorial. *Brains, Minds, and Media*, 1:1–44.
- [16] Beierlein, M., Gibson, J. R., and Connors, B. W. (2000). A network of electrically coupled interneurons drives synchronized inhibition in neocortex. *Nature neuroscience*, 3(9):904–910.
- [17] Bekolay, T., Bergstra, J., Hunsberger, E., DeWolf, T., Stewart, T. C., Rasmussen, D., Choo, X., Voelker, A., and Eliasmith, C. (2014). Nengo: a python tool for building large-scale functional brain models. *Frontiers in neuroinformatics*, 7:48.
- [18] Bhuiyan, M. A., Pallipuram, V. K., Smith, M. C., Taha, T., and Jalasutram, R. (2010). Acceleration of spiking neural networks in emerging multi-core and gpu architectures. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE.
- [19] Bisong, E. (2019). Google colab. In *Building Machine Learning and Deep Learning Models on Google Cloud Platform*, pages 59–64. Springer.
- [20] Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT’2010*, pages 177–186. Springer.
- [21] Bower, J. M. (2000). *Computational Neuroscience: Trends in Research 2000*. Elsevier.
- [22] Bower, J. M. and Beeman, D. (1998). Neural modeling with genesis. In *The Book of GENESIS*, pages 17–27. Springer.
- [23] Bower, J. M. and Beeman, D. (2012). *The book of GENESIS: exploring realistic neural models with the GEneral NEural Simulation System*. Springer Science & Business Media.
- [24] Bower, J. M., Beeman, D., and Hucka, M. (2003). The genesis simulation system. *Advances in Neural Information Processing Systems*.
- [25] Braitenberg, V. (1978). Cell assemblies in the cerebral cortex. In *Theoretical approaches to complex systems*, pages 171–188. Springer.
- [26] Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). Classification and regression trees. belmont, ca: Wadsworth. *International Group*, 432(151-166):9.
- [27] Bressler, S. L. and Menon, V. (2010). Large-scale brain networks in cognition: emerging methods and principles. *Trends in cognitive sciences*, 14(6):277–290.

- [28] Bressloff, P. C. and Cowan, J. D. (2003). A spherical model for orientation and spatial–frequency tuning in a cortical hypercolumn. *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences*, 358(1438):1643–1667.
- [29] Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J. M., Diesmann, M., Morrison, A., Goodman, P. H., Harris, F. C., et al. (2007). Simulation of networks of spiking neurons: a review of tools and strategies. *Journal of computational neuroscience*, 23(3):349–398.
- [30] Brikman, Y. (2016). Why we use terraform and not chef, puppet, ansible, saltstack, or cloudformation. Retrieved April, 24:2020.
- [31] Brock, J. D., Bruce, R. F., and Cameron, M. E. (2013). Changing the world with a raspberry pi. *Journal of Computing Sciences in Colleges*, 29(2):151–153.
- [32] Brouwer, L. E. J. (2011). *Brouwer’s Cambridge lectures on intuitionism*. Cambridge University Press.
- [33] Brown, C. (2018a). Assemble pico 3h rockpro64. <https://www.picocluster.com/blogs/picocluster-assembly-instructions/assemble-pico-3h-rockpro64/>, accessed on 21.10.2022.
- [34] Brown, E. (2018b). Sbc clusters — beyond raspberry pi.
- [35] Brown, T. (2017). Bringing docker to windows developers with windows server containers. <https://docs.microsoft.com/en-us/archive/msdn-magazine/2017/april/containers-bringing-docker-to-windows-developers-with-windows-server-containers>, accessed on 10.07.2022.
- [36] Brownlee, J. (2016). *XGBoost With python: Gradient boosted trees with XGBoost and scikit-learn*. Machine Learning Mastery.
- [37] Burns, G. A. and Young, M. P. (2000). Analysis of the connectional organization of neural systems associated with the hippocampus in rats. *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences*, 355(1393):55–70.
- [38] B.V., K. (2017). Top 7 benefits of using containers. <https://blog.kumina.nl/2017/04/the-benefits-of-containers-and-container-technology/>, accessed on 27.07.2022.
- [39] Carnevale, N. T. and Hines, M. L. (2006). *The NEURON book*. Cambridge University Press.
- [40] Chen, G. (2018). The rise of the enterprise container platform. *IDC White Paper*.
- [41] Chen, Y. and Wu, Q. (2012). Neuromorphic computing: A soc scaling path for the next decades. In *2012 IEEE International SOC Conference*, pages 290–291. IEEE.
- [42] Chlasta, K. and Wołk, K. (2021). Towards computer-based automated screening of dementia through spontaneous speech. *Frontiers in Psychology*, 11:623237.
- [43] Coop, A. D. and Reeke, G. N. (2001). The composite neuron: a realistic one-compartment purkinje cell model suitable for large-scale neuronal network simulations. *Journal of computational neuroscience*, 10(2):173–186.

- [44] Crone, J. C., Vindiola, M. M., Yu, A. B., Boothe, D. L., Beeman, D., Oie, K. S., and Franaszczuk, P. J. (2019). Enabling large-scale simulations with the genesis neuronal simulator. *Frontiers in neuroinformatics*, 13:69.
- [45] Cummings, J. L. and Cole, G. (2002). Alzheimer Disease. *JAMA*, 287(18):2335–2338.
- [46] da Cruz, S. M. S., Campos, M. L. M., and Mattoso, M. (2009). Towards a taxonomy of provenance in scientific workflow management systems. In *2009 Congress on Services-I*, pages 259–266. IEEE.
- [47] Dagum, L. and Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55.
- [48] Davies, M., Wild, A., Orchard, G., Sandamirskaya, Y., Guerra, G. A. F., Joshi, P., Plank, P., and Risbud, S. R. (2021). Advancing neuromorphic computing with loihi: A survey of results and outlook. *Proceedings of the IEEE*, 109(5):911–934.
- [49] Davison, A. P., Brüderle, D., Eppler, J. M., Kremkow, J., Müller, E., Pecevski, D., Perrinet, L., and Yger, P. (2009). Pynn: a common interface for neuronal network simulators. *Frontiers in neuroinformatics*, 2:11.
- [50] de Bayser, M. and Cerqueira, R. (2017). Integrating mpi with docker for hpc. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pages 259–265. IEEE.
- [51] De Kamps, M., Baier, V., Drever, J., Dietz, M., Mösenlechner, L., and Van Der Velde, F. (2008). The state of miind. *Neural networks*, 21(8):1164–1181.
- [52] Deng, L. (2012). The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142.
- [53] Dhamala, M., Rangarajan, G., and Ding, M. (2008). Analyzing information flow in brain networks with nonparametric granger causality. *Neuroimage*, 41(2):354–362.
- [54] Ding, S., Zhao, H., Zhang, Y., Xu, X., and Nie, R. (2015). Extreme learning machine: algorithm, theory and applications. *Artificial Intelligence Review*, 44(1):103–115.
- [55] Dobosz, K., Mikołajewski, D., Wójcik, G. M., and Duch, W. (2013). Simple cyclic movements as a distinct autism feature-computational approach. *Computer Science*, 14.
- [56] Drischel, H. and Elze, P. (1976). *Podstawy biocybernetyki*. Państwowe Wydawnictwo Naukowe.
- [57] Duch, W. (2000). Therapeutic applications of computer models of brain activity for alzheimer disease.
- [58] Duch, W. (2019). Autism spectrum disorder and deep attractors in neurodynamics. In *Multiscale Models of Brain Disorders*, pages 135–146. Springer.
- [59] Duch, W., Dobosz, K., and Mikołajewski, D. (2013). Autism and adhd—two ends of the same spectrum? In *International Conference on Neural Information Processing*, pages 623–630. Springer.

- [60] Duch, W., Nowak, W., Meller, J., Osiński, G., Dobosz, K., Mikołajewski, D., and Wójcik, G. M. (2012). Computational approach to understanding autism spectrum disorders. *Computer Science*, 13(2):47–47.
- [61] Edoctoronline (2021). Labelled side view of the human visual pathway. http://www.edoctoronline.com/media/19/photos_6F6DC7DF-28FE-433F-9AED-2CB301F35C4B.jpg, accessed on 24.11.2021.
- [62] Education, I. C. (2021). Container orchestration. <https://www.ibm.com/cloud/learn/container-orchestration>, accessed on 27.06.2022.
- [63] Eeckman, F. H. (2012). *Computation in neurons and neural systems*. Springer Science & Business Media.
- [64] Efron, B., Hastie, T., Johnstone, I., and Tibshirani, R. (2004). Least angle regression. *The Annals of statistics*, 32(2):407–499.
- [65] Einevoll, G. T., Destexhe, A., Diesmann, M., Grün, S., Jirsa, V., de Kamps, M., Migliore, M., Ness, T. V., Plesser, H. E., and Schürmann, F. (2019). The scientific case for brain simulations. *Neuron*, 102(4):735–744.
- [66] Eliasmith, C. and Trujillo, O. (2014). The use and abuse of large-scale brain models. *Current opinion in neurobiology*, 25:1–6.
- [67] Färber, F., May, N., Lehner, W., Große, P., Müller, I., Rauhe, H., and Dees, J. (2012). The sap hana database—an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33.
- [68] Felleman, D. J. and Van Essen, D. C. (1991). Distributed hierarchical processing in the primate cerebral cortex. *Cerebral cortex (New York, NY: 1991)*, 1(1):1–47.
- [69] Fenton, G. W. (1986). The eeg, epilepsy and psychiatry. *What is epilepsy*, pages 139–60.
- [70] Florescu, D. and Coca, D. (2019). Learning with Precise Spike Times: A New Decoding Algorithm for Liquid State Machines. *Neural Computation*, 31(9):1825–1852.
- [71] Fortune, S. and Wyllie, J. (1978). Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118.
- [72] Freund, Y. and Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139.
- [73] Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232.
- [74] Furber, S. B., Galluppi, F., Temple, S., and Plana, L. A. (2014). The spinnaker project. *Proceedings of the IEEE*, 102(5):652–665.
- [75] Fushiki, T. (2011). Estimation of prediction error by using k-fold cross-validation. *Statistics and Computing*, 21(2):137–146.

- [76] Fusi, S., Miller, E. K., and Rigotti, M. (2016). Why neurons mix: high dimensionality for higher cognition. *Current opinion in neurobiology*, 37:66–74.
- [77] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., et al. (2004). Open mpi: Goals, concept, and design of a next generation mpi implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 97–104. Springer.
- [78] Garey, M. R., Johnson, D. S., and Stockmeyer, L. (1974). Some simplified np-complete problems. In *Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 47–63.
- [79] Garner, A. and Mayford, M. (2012). New approaches to neural circuits in behavior. *Learning & memory*, 19(9):385–390.
- [80] Gartner (2022). Gartner says worldwide iaas public cloud services market grew 41.4% in 2021. <https://www.gartner.com/en/newsroom/press-releases/2022-06-02-gartner-says-worldwide-iaas-public-cloud-services-market-grew-41-percent-in-2021>, accessed on 02.12.2022.
- [81] Gerstner, W. and Kistler, W. M. (2002). *Spiking neuron models: Single neurons, populations, plasticity*. Cambridge university press.
- [82] Gewaltig, M.-O. and Diesmann, M. (2007). Nest (neural simulation tool). *Scholarpedia*, 2(4):1430.
- [83] Gholampour, S. and Fatourae, N. (2021). Boundary conditions investigation to improve computer simulation of cerebrospinal fluid dynamics in hydrocephalus patients. *Communications biology*, 4(1):1–15.
- [84] Gigerenzer, G. and Goldstein, D. G. (1996). Mind as computer: Birth of a metaphor. *Creativity Research Journal*, 9(2-3):131–144.
- [85] Gneiting, T. and Walz, E.-M. (2022). Receiver operating characteristic (roc) movies, universal roc (uroc) curves, and coefficient of predictive ability (cpa). *Machine Learning*, 111(8):2769–2797.
- [86] Goddard, N. H. and Hood, G. (1997). Parallel genesis for large-scale modeling. In *Computational Neuroscience*, pages 911–917. Springer.
- [87] Goodman, D. F. and Brette, R. (2008). Brian: a simulator for spiking neural networks in python. *Frontiers in neuroinformatics*, page 5.
- [88] Grzyb, B. J., Chinellato, E., Wojcik, G. M., and Kaminski, W. A. (2009). Which model to use for the liquid state machine? In *2009 International Joint Conference on Neural Networks*, pages 1018–1024. IEEE.
- [89] Gupta, A., Wang, Y., and Markram, H. (2000). Organizing principles for a diversity of gabaergic interneurons and synapses in the neocortex. *Science*, 287(5451):273–278.
- [90] Habib, S., Roser, R., Gerber, R., Antypas, K., Riley, K., Williams, T., Wells, J., Straatsma, T., Almgren, A., Amundson, J., et al. (2016). Asc/hep exascale requirements review report. *arXiv preprint arXiv:1603.09303*.

- [91] Halan, D. (2017). The blue brain project: Unraveling the brain's mystery. <https://www.electronicsforu.com/technology-trends/must-read/blue-brain-project-unraveling-brains-mystery/2>, accessed on 03.02.2022.
- [92] Hale, J. S., Li, L., Richardson, C. N., and Wells, G. N. (2017). Containers for portable, productive, and performant scientific computing. *Computing in Science & Engineering*, 19(6):40–50.
- [93] Harris, D. C. (1998). Nonlinear least-squares curve fitting with microsoft excel solver. *Journal of chemical education*, 75(1):119.
- [94] Hasani, R., Lechner, M., Amini, A., Liebenwein, L., Ray, A., Tschalkowski, M., Teschl, G., and Rus, D. (2022). Closed-form continuous-time neural networks. *Nature Machine Intelligence*, pages 1–12.
- [95] Hastie, T., Rosset, S., Zhu, J., and Zou, H. (2009). Multi-class adaboost. *Statistics and its Interface*, 2(3):349–360.
- [96] Hazan, H. and Manevitz, L. M. (2012). Topological constraints and robustness in liquid state machines. *Expert Systems with Applications*, 39(2):1597–1606.
- [97] Hebb, D. O. (2005). *The organization of behavior: A neuropsychological theory*. Psychology Press.
- [98] Helias, M., Kunkel, S., Masumoto, G., Igarashi, J., Eppler, J. M., Ishii, S., Fukai, T., Morrison, A., and Diesmann, M. (2012). Supercomputers ready for use as discovery machines for neuroscience. *Frontiers in neuroinformatics*, 6:26.
- [99] Hepburn, I., Chen, W., Wils, S., and De Schutter, E. (2012). Steps: efficient simulation of stochastic reaction–diffusion models in realistic morphologies. *BMC systems biology*, 6(1):1–19.
- [100] Herculano-Houzel, S. (2009). The human brain in numbers: a linearly scaled-up primate brain. *Frontiers in human neuroscience*, 3:31.
- [101] Hill, S. and Markram, H. (2008). The blue brain project. In *2008 30th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages clviii–clviii. IEEE.
- [102] Hines, M., Kumar, S., and Schürmann, F. (2011). Comparison of neuronal spike exchange methods on a blue gene/p supercomputer. *Frontiers in computational neuroscience*, 5:49.
- [103] Hines, M. L. and Carnevale, N. T. (2001). Neuron: a tool for neuroscientists. *The neuroscientist*, 7(2):123–135.
- [104] Hines, M. L., Morse, T., Migliore, M., Carnevale, N. T., and Shepherd, G. M. (2004). Modeldb: a database to support computational neuroscience. *Journal of computational neuroscience*, 17(1):7–11.
- [105] Hinton, G. E. (1990). Connectionist learning procedures. In *Machine learning*, pages 555–610. Elsevier.

- [106] Hirota, T. and King, B. H. (2023). Autism Spectrum Disorder: A Review. *JAMA*, 329(2):157–168.
- [107] Hodgkin, A. L. and Huxley, A. F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117(4):500–544.
- [108] Holden, A., Tucker, J., and Thompson, B. (1991). Can excitable media be considered as computational systems? *Physica D: Nonlinear Phenomena*, 49(1-2):240–246.
- [109] Holmes, L. (2012). *Windows PowerShell Cookbook: The Complete Guide to Scripting Microsoft's Command Shell*. O'Reilly Media.
- [110] Hopfield, J. J. and Brody, C. D. (2000). What is a moment? “cortical” sensory integration over a brief interval. *Proceedings of the National Academy of Sciences*, 97(25):13919–13924.
- [111] Huang, G.-B., Zhu, Q.-Y., and Siew, C.-K. (2004). Extreme learning machine: a new learning scheme of feedforward neural networks. In *2004 IEEE international joint conference on neural networks (IEEE Cat. No. 04CH37541)*, volume 2, pages 985–990. Ieee.
- [112] Huang, G.-B., Zhu, Q.-Y., and Siew, C.-K. (2006). Extreme learning machine: theory and applications. *Neurocomputing*, 70(1-3):489–501.
- [113] Huang, J., Wang, Y., and Huang, J. (2009). The separation property enhancement of liquid state machine by particle swarm optimization. In *International Symposium on Neural Networks*, pages 67–76. Springer.
- [114] Idoine, C., Krensky, P., Brethenoux, E., Hare, J., Sicular, S., and Vashisth, S. (2018). Magic quadrant for data science and machine-learning platforms. *Gartner, Inc*, page 13.
- [115] Inc., P. M. (2022). Rockpro64.
- [116] Ippen, T., Eppler, J. M., Plessner, H. E., and Diesmann, M. (2017). Constructing neuronal network models in massively parallel environments. *Frontiers in neuroinformatics*, 11:30.
- [117] Ivanov, V. and Michmizos, K. (2021). Increasing liquid state machine performance with edge-of-chaos dynamics organized by astrocyte-modulated plasticity. *Advances in Neural Information Processing Systems*, 34.
- [118] Izhikevich, E. M. (2003). Simple model of spiking neurons. *IEEE Transactions on neural networks*, 14(6):1569–1572.
- [119] Izhikevich, E. M. (2004). Which model to use for cortical spiking neurons? *IEEE transactions on neural networks*, 15(5):1063–1070.
- [120] Jaeger, H. (2001). The “echo state” approach to analysing and training recurrent neural networks—with an erratum note. *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, 148(34):13.

- [121] Jaeger, H., Lukoševičius, M., Popovici, D., and Siewert, U. (2007a). Optimization and applications of echo state networks with leaky-integrator neurons. *Neural networks*, 20(3):335–352.
- [122] Jaeger, H., Maass, W., and Principe, J. (2007b). Special issue on echo state networks and liquid state machines.
- [123] Jahankhani, P., Revett, K., and Kodogiannis, V. (2005). Detecting clinically relevant eeg anomalies using discrete wavelet transforms. WSEAS.
- [124] Johnston, S. J., Basford, P. J., Perkins, C. S., Herry, H., Tso, F. P., Pezaros, D., Mullins, R. D., Yoneki, E., Cox, S. J., and Singer, J. (2018). Commodity single board computer clusters and their applications. *Future Generation Computer Systems*, 89:201–212.
- [125] Jordan, J., Ippen, T., Helias, M., Kitayama, I., Sato, M., Igarashi, J., Diesmann, M., and Kunkel, S. (2018). Extremely scalable spiking neuronal network simulation code: from laptops to exascale computers. *Frontiers in neuroinformatics*, 12:2.
- [126] Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T.-Y. (2017). Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, 30.
- [127] Khaydarova, R., Fishchenko, V., Mouromtsev, D., Shmatkov, V., and Lapaev, M. (2020). Rock-cnn: a distributed rockpro64-based convolutional neural network cluster for iot. verification and performance analysis. In *2020 26th Conference of Open Innovations Association (FRUCT)*, pages 174–181. IEEE.
- [128] Khaydarova, R., Mouromtsev, D., Fishchenko, V., Shmatkov, V., Lapaev, M., and Shilin, I. (2021). Rock-cnn: distributed deep learning computations in a resource-constrained cluster. *International Journal of Embedded and Real-Time Communication Systems (IJERTCS)*, 12(3):14–31.
- [129] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.
- [130] Kirschstein, T. and Köhling, R. (2009). What is the source of the eeg? *Clinical EEG and neuroscience*, 40(3):146–149.
- [131] Knuth, D. E. (1976). Big omicron and big omega and big theta. *ACM Sigact News*, 8(2):18–24.
- [132] Kong, Q., Siau, T., and Bayen, A. (2020). *Python programming and numerical methods: A guide for engineers and scientists*. Academic Press.
- [133] Konkoli, Z. (2017). On reservoir computing: from mathematical foundations to unconventional applications. In *Advances in unconventional computing*, pages 573–607. Springer.
- [134] Konkoli, Z., Nichele, S., Dale, M., and Stepney, S. (2018). Reservoir computing with computational matter. In *Computational Matter*, pages 269–293. Springer.

- [135] Kotipalli, P. V., Singh, R., Wood, P., Laguna, I., and Bagchi, S. (2019). Ampt-ga: automatic mixed precision floating point tuning for gpu applications. In *Proceedings of the ACM International Conference on Supercomputing*, pages 160–170.
- [136] Kumar, M., Mishra, S., Lathar, N. K., and Singh, P. (2023). Infrastructure as code (iac): Insights on various platforms. In *Sentiment Analysis and Deep Learning: Proceedings of ICSADL 2022*, pages 439–449. Springer.
- [137] Kumbhar, P., Hines, M., Fouriaux, J., Ovcharenko, A., King, J., Delalondre, F., and Schürmann, F. (2019). Coreuron: an optimized compute engine for the neuron simulator. *Frontiers in neuroinformatics*, 13:63.
- [138] Kunkel, S., Potjans, T. C., Eppler, J. M., Plesser, H. E. E., Morrison, A., and Diesmann, M. (2012). Meeting the memory challenges of brain-scale network simulation. *Frontiers in neuroinformatics*, 5:35.
- [139] Landhuis, E. (2017). Neuroscience: Big brain, big data. *Nature*, 541(7638):559–561.
- [140] Landrigan, P. J. (2010). What causes autism? exploring the environmental contribution. *Current opinion in pediatrics*, 22(2):219–225.
- [141] Lent, R., Azevedo, F. A., Andrade-Moraes, C. H., and Pinto, A. V. (2012). How many neurons do you have? some dogmas of quantitative neuroscience under revision. *European Journal of Neuroscience*, 35(1):1–9.
- [142] Loporati, A., Zandron, C., Ferretti, C., and Mauri, G. (2007). Solving numerical np-complete problems with spiking neural p systems. In *Membrane Computing: 8th International Workshop, WMC 2007 Thessaloniki, Greece, June 25-28, 2007 Revised Selected and Invited Papers 8*, pages 336–352. Springer.
- [143] Lever, J. (2016). Classification evaluation: It is important to understand both what a classification metric expresses and what it hides. *Nature methods*, 13(8):603–605.
- [144] Llinas, R. and Ribary, U. (1993). Coherent 40-hz oscillation characterizes dream state in humans. *Proceedings of the National Academy of Sciences*, 90(5):2078–2081.
- [145] Lopes da Silva, F. and Niedermeyer, E. (2005). Electroencephalography, basic principles, clinical applications and related fields. *5th edition*.
- [146] Maass, W. (1996). Lower bounds for the computational power of networks of spiking neurons. *Neural computation*, 8(1):1–40.
- [147] Maass, W. (2011). Liquid state machines: motivation, theory, and applications. *Computability in context: computation and logic in the real world*, pages 275–296.
- [148] Maass, W., Legenstein, R., and Bertschinger, N. (2004a). Methods for estimating the computational power and generalization capability of neural microcircuits. *Advances in neural information processing systems*, 17.
- [149] Maass, W. and Markram, H. (2004). On the computational power of circuits of spiking neurons. *Journal of computer and system sciences*, 69(4):593–616.

- [150] Maass, W., Natschläger, T., and Markram, H. (2002a). A model for real-time computation in generic neural microcircuits. *Advances in neural information processing systems*, 15.
- [151] Maass, W., Natschläger, T., and Markram, H. (2002b). Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural computation*, 14(11):2531–2560.
- [152] Maass, W., Natschläger, T., and Markram, H. (2004b). Computational models for generic cortical microcircuits. *Computational neuroscience: A comprehensive approach*, 18:575–605.
- [153] Maass, W. and Sontag, E. D. (2000). Neural systems as nonlinear filters. *Neural computation*, 12(8):1743–1772.
- [154] Maksimović, M., Vujović, V., Davidović, N., Milošević, V., and Perišić, B. (2014). Raspberry pi as internet of things hardware: performances and constraints. *design issues*, 3(8):1–6.
- [155] Markram, H. (2006). The blue brain project. *Nature Reviews Neuroscience*, 7(2):153–160.
- [156] Markram, H., Muller, E., Ramaswamy, S., Reimann, M. W., Abdellah, M., Sanchez, C. A., Ailamaki, A., Alonso-Nanclares, L., Antille, N., Arsever, S., et al. (2015). Reconstruction and simulation of neocortical microcircuitry. *Cell*, 163(2):456–492.
- [157] Martin, K. A. (2002). Microcircuits in visual cortex. *Current opinion in neurobiology*, 12(4):418–425.
- [158] Martone, M., Thor, A., and Price, D. (2002). Purkinje neuron from mouse cerebellum injected with lucifer yellow and imaged using confocal microscopy. <https://library.ucsd.edu/dc/object/bb03097189>, accessed on 09.09.2021.
- [159] McCormick, D. A., Shu, Y., and Yu, Y. (2007). Hodgkin and huxley model—still standing? *Nature*, 445(7123):E1–E2.
- [160] McDaniel-Gray, S., Boothe, D., Yu, A., Shires, D., and Taufer, M. (2018). Leveraging in situ data analysis to enable computational steering of brain’s neocortex simulations with genesis. In *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 873–880. IEEE.
- [161] Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., et al. (2016). Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241.
- [162] Merkel, D. et al. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux j*, 239(2):2.

- [163] Migliore, M., Cannia, C., Lytton, W. W., Markram, H., and Hines, M. L. (2006). Parallel network simulations with neuron. *Journal of computational neuroscience*, 21(2):119–129.
- [164] Migliore, M., Morse, T. M., Davison, A. P., Marenco, L., Shepherd, G. M., and Hines, M. L. (2003). Modeldb.
- [165] Mikołajewska, E. and Mikołajewski, D. (2011). Zastosowanie medyczne systemów ambient intelligence. *Acta Bio-Optica et Informatica Medica. Inżynieria Biomedyczna*, 17(3):207–210.
- [166] Miller, G. A. (2003). The cognitive revolution: a historical perspective. *Trends in cognitive sciences*, 7(3):141–144.
- [167] Monroe, D. (2014). Neuromorphic computing gets ready for the (really) big time.
- [168] Morales, A. W. (2021). Why oracle engineers love raspberry pi projects.
- [169] Mott, M. C., Gordon, J. A., and Koroshetz, W. J. (2018). The nih brain initiative: Advancing neurotechnologies, integrating disciplines. *PLoS biology*, 16(11):e3000066.
- [170] Mountcastle, V. B. (1997). The columnar organization of the neocortex. *Brain: a journal of neurology*, 120(4):701–722.
- [171] Musilova, J. and Sedlar, K. (2021). Tools for time-course simulation in systems biology: a brief overview. *Briefings in Bioinformatics*, 22(5):bbaa392.
- [172] Naik, N. (2022). Cloud-agnostic and lightweight big data processing platform in multiple clouds using docker swarm and terraform. In *UK Workshop on Computational Intelligence*, pages 519–531. Springer.
- [173] Natschläger, T., Maass, W., and Markram, H. (2002). The "liquid computer": A novel strategy for real-time computing on time series. *Special issue on Foundations of Information Processing of TELEMATIK*, 8(ARTICLE):39–43.
- [174] Nickoloff, J. and Kuenzli, S. (2019). *Docker in action*. Simon and Schuster.
- [175] Noback, C. R., Ruggiero, D. A., Strominger, N. L., and Demarest, R. J. (2005). *The human nervous system: structure and function*. Number 744. Springer Science & Business Media.
- [176] Norton, D. and Ventura, D. (2009). Improving the separability of a reservoir facilitates learning transfer. In *2009 International Joint Conference on Neural Networks*, pages 2288–2293. IEEE.
- [177] Norton, D. and Ventura, D. (2010). Improving liquid state machines through iterative refinement of the reservoir. *Neurocomputing*, 73(16-18):2893–2904.
- [178] Orchard, G., Jayawant, A., Cohen, G. K., and Thakor, N. (2015). Converting static image datasets to spiking neuromorphic datasets using saccades. *Frontiers in neuroscience*, 9:437.

- [179] O’reilly, R. C. and Munakata, Y. (2000). *Computational explorations in cognitive neuroscience: Understanding the mind by simulating the brain*. MIT press.
- [180] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.
- [181] Piccinini, G. and Bahar, S. (2013). Neural computation and the computational theory of cognition. *Cognitive science*, 37(3):453–488.
- [182] Pinker, S. (2005). So how does the mind work? *Mind & language*, 20(1):1–24.
- [183] Plesser, H. E., Eppler, J. M., Morrison, A., Diesmann, M., and Gewaltig, M.-O. (2007). Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers. In *European Conference on Parallel Processing*, pages 672–681. Springer.
- [184] Purves, D., Augustine, G., Fitzpatrick, D., Katz, L., LaMantia, A., McNamara, J., and Mark Williams, S. (2001). Excitatory and inhibitory postsynaptic potentials. *Neuroscience*, pages 2–3.
- [185] Ramey, C. (1994). Bash, the bourne- again shell. In *Proceedings of The Romanian Open Systems Conference & Exhibition (ROSE 1994), The Romanian UNIX User’s Group (GURU)*, pages 3–5.
- [186] Rescorla, M. (2015). The computational theory of mind.
- [187] Ritter, P., Schirner, M., McIntosh, A. R., and Jirsa, V. K. (2013). The virtual brain integrates computational modeling and multimodal neuroimaging. *Brain connectivity*, 3(2):121–145.
- [188] Rubinov, M. and Sporns, O. (2010). Complex network measures of brain connectivity: uses and interpretations. *Neuroimage*, 52(3):1059–1069.
- [189] Scannell, J. W., Blakemore, C., and Young, M. P. (1995). Analysis of connectivity in the cat cerebral cortex. *Journal of Neuroscience*, 15(2):1463–1483.
- [190] Schölkopf, B., Smola, A. J., Williamson, R. C., and Bartlett, P. L. (2000). New support vector algorithms. *Neural computation*, 12(5):1207–1245.
- [191] Schuman, C. D., Kulkarni, S. R., Parsa, M., Mitchell, J. P., Kay, B., et al. (2022). Opportunities for neuromorphic computing algorithms and applications. *Nature Computational Science*, 2(1):10–19.
- [192] Shannon, C. E. (1956). A universal turing machine with two internal states. *Automata studies*, 34:157–165.
- [193] Shehzad, D. and Bozkus, Z. (2015). Optimizing neuron brain simulator with remote memory access on distributed memory systems. In *2015 International Conference on Emerging Technologies (ICET)*, pages 1–5. IEEE.
- [194] Shepherd, G. M. (1988). *A basic circuit for cortical organization*. MIT-Press.

- [195] Shi, Y.-L., Steinmetz, N. A., Moore, T., Boahen, K., and Engel, T. A. (2022). Cortical state dynamics and selective attention define the spatial pattern of correlated variability in neocortex. *Nature communications*, 13(1):1–15.
- [196] Simpson, S. L., Bowman, F. D., and Laurienti, P. J. (2013). Analyzing complex functional brain networks: fusing statistics and network science to understand the brain. *Statistics surveys*, 7:1.
- [197] Smith, M. R., Hill, A. J., Carlson, K. D., Vineyard, C. M., Donaldson, J., Follett, D. R., Follett, P. L., Naegle, J. H., James, C. D., and Aimone, J. B. (2017). A novel digital neuromorphic architecture efficiently facilitating complex synaptic response functions applied to liquid state machines. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 2421–2428. IEEE.
- [198] Sobota, J., Goubej, M., Königsmarková, J., and Čech, M. (2019). Raspberry pi-based hil simulators for control education. *IFAC-PapersOnLine*, 52(9):68–73.
- [199] Soni, M. (2015). End to end automation on cloud with build pipeline: the case for devops in insurance industry, continuous integration, continuous testing, and continuous delivery. In *2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pages 85–89. IEEE.
- [200] Sporns, O., Chialvo, D. R., Kaiser, M., and Hilgetag, C. C. (2004). Organization, development and function of complex brain networks. *Trends in cognitive sciences*, 8(9):418–425.
- [201] Sterratt, D., Graham, B., Gillies, A., and Willshaw, D. (2011). *Principles of computational modelling in neuroscience*. Cambridge University Press.
- [202] Swanson, L. W. and Bota, M. (2010). Foundational model of structural connectivity in the nervous system with a schema for wiring diagrams, connectome, and basic plan architecture. *Proceedings of the National Academy of Sciences*, 107(48):20610–20617.
- [203] Tadeusiewicz, R. (1993). *Sieci neuronowe*, volume 110. Akademicka Oficyna Wydawnicza RM Warszawa.
- [204] Tadeusiewicz, R. (2010a). New trends in neurocybernetics. *Computer Methods in Materials Science*, 10(1):1–7.
- [205] Tadeusiewicz, R. (2010b). Using neural networks for simplified discovery of some psychological phenomena. In *International Conference on Artificial Intelligence and Soft Computing*, pages 104–123. Springer.
- [206] Tadeusiewicz, R. (2015). Neural networks as a tool for modeling of biological systems. *Bio-Algorithms and Med-Systems*, 11(3):135–144.
- [207] Tadeusiewicz, R. (2018). Introduction to intelligent systems. In *Intelligent systems*, pages 1–1. CRC Press.
- [208] Tadeusiewicz, R. and Ogiela, M. (2005). New proposition for intelligent systems design: artificial understanding of the images as the next step of advanced data analysis after automatic classification and pattern recognition. In *5th International Conference on Intelligent Systems Design and Applications (ISDA '05)*, pages 297–300.

- [209] Tadeusiewicz, R. and Ogiela, M. R. (2003). Artificial intelligence techniques in retrieval of visual data semantic information. In *Advances in Web Intelligence: First International AtlanticWeb Intelligence Conference, AWIC 2003, Madrid, Spain, May 5–6, 2003. Proceedings 1*, pages 18–27. Springer.
- [210] Tadeusiewicz, R., Śmiałowska, M., Hess, G., Błaszczuk, J., Kamiński, W., Lazarewicz, M., et al. (2009). *Neurocybernetyka teoretyczna*. Wydawnictwa Uniwersytetu Warszawskiego Warsaw.
- [211] Tadeusiewicz, R., Tylek, P., Adamczyk, F., Kiełbasa, P., Jabłoński, M., Bubleński, Z., Grabska-Chrzastowska, J., Kaliniewicz, Z., Walczyk, J., Szczepaniak, J., et al. (2017). Assessment of selected parameters of the automatic scarification device as an example of a device for sustainable forest management. *Sustainability*, 9(12):2370.
- [212] Tadeusiewicz, R. and Wajs, W. (1999). *Informatyka medyczna*. Uczelniane Wydawnictwa Naukowo-Dydaktyczne Akademii Górniczo-Hutniczej.
- [213] Tamraz, J. C., Comair, Y. G., and Tamraz, J. (2004). *Atlas of regional anatomy of the brain using MRI*. Springer.
- [214] Tausen, M., Clausen, M., Moeskjær, S., Shihavuddin, A., Dahl, A. B., Janss, L., and Andersen, S. U. (2020). Greenotyper: Image-based plant phenotyping using distributed computing and deep learning. *Frontiers in plant science*, 11:1181.
- [215] Tikidji-Hamburyan, R. A., Narayana, V., Bozkus, Z., and El-Ghazawi, T. A. (2017). Software for brain network simulations: a comparative study. *Frontiers in neuroinformatics*, 11:46.
- [216] Tong, F. (2003). Primary visual cortex and visual awareness. *Nature Reviews Neuroscience*, 4(3):219–229.
- [217] Tso, F. P., White, D. R., Jouet, S., Singer, J., and Pezaros, D. P. (2013). The glasgow raspberry pi cloud: A scale model for cloud computing infrastructures. In *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops*, pages 108–112. IEEE.
- [218] Tsunoda, K., Yamane, Y., Nishizaki, M., and Tanifuji, M. (2001). Complex objects are represented in macaque inferotemporal cortex by the combination of feature columns. *Nature neuroscience*, 4(8):832–838.
- [219] Ursuțiu, D., Nascov, V., Samoilă, C., and Moga, M. (2012). Microcontroller technologies in low power applications. In *2012 15th International Conference on Interactive Collaborative Learning (ICL)*, pages 1–5. IEEE.
- [220] Vayttaden, S. J. and Bhalla, U. S. (2004). Developing complex signaling models using genesis/kinetikit. *Science's STKE*, 2004(219):pl4–pl4.
- [221] Vujović, V. and Maksimović, M. (2014). Raspberry pi as a wireless sensor node: Performances and constraints. In *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1013–1018. IEEE.

- [222] Waldrop, M. (2012). Computer modelling: Brain in a box. *Nature News*, 482(7386):456.
- [223] Wang, L., Yang, Z., Guo, S., Qu, L., Zhang, X., Kang, Z., and Xu, W. (2022). Lsmcore: A 69k-synapse/mm² single-core digital neuromorphic processor for liquid state machine. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 69(5):1976–1989.
- [224] Weuve, J., Hebert, L. E., Scherr, P. A., and Evans, D. A. (2015). Prevalence of alzheimer disease in us states. *Epidemiology*, 26(1):e4–e6.
- [225] White, J. G., Southgate, E., Thomson, J. N., Brenner, S., et al. (1986). The structure of the nervous system of the nematode *caenorhabditis elegans*. *Philos Trans R Soc Lond B Biol Sci*, 314(1165):1–340.
- [226] Wijesinghe, P., Srinivasan, G., Panda, P., and Roy, K. (2019). Analysis of liquid ensembles for enhancing the performance and accuracy of liquid state machines. *Frontiers in neuroscience*, 13:504.
- [227] Wojcik, G. M. (2012). Self-organising criticality in the simulated models of the rat cortical microcircuits. *Neurocomputing*, 79:61–67.
- [228] Wojcik, G. M. and Garcia-Lazaro, J. A. (2010). Analysis of the neural hypercolumn in parallel pcsim simulations. *Procedia Computer Science*, 1(1):845–854.
- [229] Wojcik, G. M. and Kaminski, W. A. (2004). Liquid state machine built of hodgkin–huxley neurons and pattern recognition. *Neurocomputing*, 58:245–251.
- [230] Wojcik, G. M. and Kaminski, W. A. (2006). Liquid computations and large simulations of the mammalian visual cortex. In *International Conference on Computational Science*, pages 94–101. Springer.
- [231] Wojcik, G. M. and Kaminski, W. A. (2007). Liquid state machine and its separation ability as function of electrical parameters of cell. *Neurocomputing*, 70(13-15):2593–2597.
- [232] Wojcik, G. M. and Ważny, M. (2015). Bray-curtis metrics as measure of liquid state machine separation ability in function of connections density. *Procedia Computer Science*, 51:2948–2951.
- [233] Xiao, H., Rasul, K., and Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms.
- [234] Yamanoor, N. S. and Yamanoor, S. (2017). High quality, low cost education with the raspberry pi. In *2017 IEEE Global Humanitarian Technology Conference (GHTC)*, pages 1–5. IEEE.
- [235] Yamazaki, T. and Tanaka, S. (2007). The cerebellum as a liquid state machine. *Neural Networks*, 20(3):290–297.
- [236] Yavuz, E., Turner, J., and Nowotny, T. (2016). Genn: a code generation framework for accelerated brain simulations. *Scientific reports*, 6(1):1–14.
- [237] Yong, E. (2019). The human brain project hasn’t lived up to its promise. *the atlantic*.

- [238] Yoo, A. B., Jette, M. A., and Grondona, M. (2003). Slurm: Simple linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*, pages 44–60. Springer.
- [239] Yorozu, T., Hirano, M., Oka, K., and Tagawa, Y. (1987). Electron spectroscopy studies on magneto-optical media and plastic substrate interface. *IEEE translation journal on magnetism in Japan*, 2(8):740–741.
- [240] Yu, H.-F., Huang, F.-L., and Lin, C.-J. (2011). Dual coordinate descent methods for logistic regression and maximum entropy models. *Machine Learning*, 85(1):41–75.
- [241] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 15–28.
- [242] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., Stoica, I., et al. (2010). Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95.
- [243] Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., and Stoica, I. (2013). Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 423–438.
- [244] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., et al. (2016). Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65.
- [245] Zenke, F. and Nefci, E. O. (2021). Brain-inspired learning on neuromorphic substrates. *Proceedings of the IEEE*, 109(5):935–950.
- [246] Zhou, Z., Chen, Y., Ding, M., Wright, P., Lu, Z., and Liu, Y. (2009). Analyzing brain networks with pca and conditional granger causality. *Human brain mapping*, 30(7):2197–2206.
- [247] École Polytechnique Fédérale de Lausanne (2022). Frequently asked questions about the blue brain project. https://www.epfl.ch/research/domains/bluebrain/frequently_asked_questions/, accessed on 06.03.2022.

Appendix A

Academic & Professional Profile

Contact: chlasta@pja.edu.pl, +48 572 885 704

Google Scholar: <https://scholar.google.com/citations?user=uaK9co0AAAAJ>

A.1 Karol Chlasta's Resumé

Karol Chlasta is an information technology (IT) and research and development (R&D) specialist with a collaboration track with a few Polish and European universities, including the Polish-Japanese Academy of Information Technology, SWPS University, Kozminski University, Warsaw University, and Hasso-Platter Institute (part of the University of Potsdam, Germany).

He is skilled in programming and shell scripting, understands systems architecture incl. cloud services. He possesses data analysis skills using a few business intelligence platforms, and data engineering skills using a few big-data technologies. He is also experienced in project, security and team management.

The PhD candidate created one of the first computer networks providing Internet access in Tarnów in the mid-90s (last century). Fan of Linux, single board computers, owner of a few vintage IBM PS/2 computers and a passionate marathon runner.

His research interests focus on intelligent systems, and application of information technology methods (in practice often different types of artificial neural networks) into important societal problems (social informatics). Since 2021 these interests started evolving towards neuroinformatics and biomedical engineering (e.g. spiking neural networks and Liquid State Machines).

A.2 Selected Formal Education

- 2018-now: Polish-Japanese Academy of Information Technology, Computer Science at ICT & Psychology Doctoral Programme
- 2022: University College Dublin, Postgraduate, University Teaching & Learning at School of Academic Affairs
- 2014-15: Warsaw University of Technology, Postgraduate, Business Analytics Systems at Institute of Computer Science, Faculty of Electronics and Information Technology
- 2003-08: Cracow University of Economics, Master of Science, Economic Computer Science at Institute of Computer Science, Faculty of Management

A.3 Professional Experience in IT Sector

- 2022-now: Technicus, Head of Technicus Poland, CISO for Technicus UK and Poland
- 2016-22: Aviva, Senior Information Technology Manager
- 2015-16: Brown-Brothers Harriman, Development Manager
- 2011-15: Royal Bank of Scotland, Senior Analyst, Team Leader, Manager
- 2009-10: MicroStrategy, Business Intelligence Architect
- 2007-09: IBM, Junior System Administrator, IT Applications Support Specialist
- 2006: Hewlett-Packard, Software Developer

A.4 Selected IT Certificates

- Huawei Certified Academy Instructor (HCAI), Huawei HCIA - Artificial Intelligence (1 exam), Huawei Cyber Security Certificate for the Subcontractor Resources (1 exam)
- MicroStrategy Certified Engineer (4 exams + project), IBM Certified Solution Expert - Cognos Business Intelligence (4 exams), IBM Certified Database Administrator - DB2 for Linux, UNIX and Windows (2 exams), SAP Certified Application Associate - SAP BusinessObjects Enterprise (1 exam), EMC Captiva Accredited InputAccel System Administrator & Developer (3 exams)

- Agile Project Management Foundation (1 exam), PRINCE2 Foundations (1 exam), Certificate of LEAN Competency (1 exam)

A.5 Academic Experience & Achievements

Since 2018 in the process of building a long-term career in academia: 19 conference presentations, 13 different publications, including 2 in-review, 3 accepted for publication and 8 published. Conducted research in applied social and neuroinformatics independently, and as part of a few research projects listed below.

Academic Collaboration

- 2021-23: Future SOC Lab (Service-Oriented Computing), Hasso-Plattner Institut, Germany. Collaborating with prof. Andreas Polze and his team in the areas of in-memory computing, cloud computing, and non-CPU (GPU, FPGA) computing:
 - Received a computational grant for LSM simulations; now responsible for a 1.5 year research project “Exploring Spiking Neural Networks for Real-Time Information Processing” (Project 125).
- 2018-23: Eye Tracking Research Center, SWPS University, Warsaw, Poland. Collaborating with prof. Izabela Krejtz and dr Krzysztof Krejtz and their teams in the fields of psychology, eye-tracking, and human-computer interaction:
 - Participated in the VXSlate research project funded by MediaFutures partners and the Research Council of Norway (grant number 309339) and Wallenberg AI, Autonomous Systems and Software Program – Humanities and Society (WASP-HS) programme at Marianne and Marcus Wallenberg Foundation.
- 2021-22: CRASH Centre for Social Change and Human Mobility, Kozminski University, Warsaw, Poland. Collaborating with prof. Izabela Grabowska and her team in the BigMig research project:
 - National Science Centre Scholarship in the research project “BigMig: Digital and non-digital traces of migrants in Big and Small Data approaches to human capacities” (OPUS-19, No. 2020/37/B/HS6/02342-1/947/2021), research completed in June 2022.
- 2020-21: Mobility Research Centre, SWPS University, Warsaw, Poland. Collaborating with dr Olga Czeranowska and her team in the (IT)mobility research project:

- Researcher in (IT)mobility project financed by the Ministry of Science and Higher Education in Poland under the 2019-2022 program called “Regional Initiative of Excellence”, project number 012/RID/2018/19. Research completed in mid-2021.

Selected Conference Presentations and Publications

- FedCSIS 2022 – 17th Conference on Computer Science and Intelligence Systems, Sofia, Bulgaria (September 4th-7th, 2022). Presented during the 1st Workshop on Personalization and Recommender Systems (PeRS’22): “MyMigrationBot: A Cloud-based Facebook Social Chatbot for Migrant Populations”
- PACIS 2021 – The Pacific Asia Conference on Information Systems, Dubai, UAE (July 12th-14th, 2021). Presented: “Hybrid approach to detecting symptoms of depression in social media entries”
- MCSB 2021 – International Conference on Cybernetic Modelling of Biological Systems, Cracow, Poland (May 28th-30th, 2021). Presented: “Liquid State Machines in parallel simulations of mammalian visual system on Raspberry Pi”
- IEEE VR 2021 – Conference on Virtual Reality and 3D User Interfaces, Lisbon, Portugal (March 27th-April 1st, 2021). Presented: “VXSlate: Combining Head Movement and Mobile Touch for Large Virtual Display Interaction”
- HCist 2019 – International Conference on Health and Social Care Information Systems and Technologies, an Association for Information Systems affiliated conference, Sousse, Tunisia (October 16-18th, 2019). Presented: “Automated speech-based screening of depression using deep convolutional neural networks”

Other Academic Experience

- 2019-23: Polish-Japanese Academy of Information Technology, Contracted Lecturer for Advanced Multimedia Techniques and Introduction to Machine Learning
- 2021-23: Kozminski University, Research and Teaching Assistant focusing on technical modules of Management & Artificial Intelligence programme
- 2020-21: University of Information Technology and Management, Olsztyn, Poland. Designed, developed, and delivered three courses for the engineering students in computer science and industrial management. Contracted Lecturer for Business Intelligence Systems, Modern Database Systems and Multimedia Technologies.

Table A.1 Author's selected scientific peer-reviewed publications.

No	Title	Publisher	Details	Points
1	Neural Simulation Pipeline: Enabling Container-based Simulations On-Premise and in Public Clouds.	Frontiers in Neuroinformatics (2023)	Volume 17, ISSN 1662-5196	140
2	MyMigrationBot: A Cloud-based Facebook Social Chatbot for Migrant Populations	FedCSIS - 17th Conference on Computer Science and Intelligence Systems (2022)	Annals of Computer Science and Information Systems ISSN 2300-5963	70
3	Liquid State Machines for Real-Time Neural Simulations	Selected Topics in Applied Computer Science, Maria Curie-Skłodowska University Press (2021)	ISBN 978-83-227-9530-9	100
4	Hybrid approach to detecting symptoms of depression in social media entries	PACIS - The Pacific Asia Conference on Information Systems (2021)	PACIS 2021 Proceedings. 192	140
5	VXSlate: Exploring Combined Head Movement and Mobile Touch for Large Virtual Display Interaction	ACM Designing Interactive Systems Conference (2021)	https://dl.acm.org/doi/10.1145/3461778.3462076	70
6	Towards computer-based automated screening of dementia through spontaneous speech	Frontiers in Psychology - Human Media Interaction (2020)	https://doi.org/10.3389/fpsyg.2020.623237	70
7	Automated speech-based screening of depression using deep convolutional neural networks	Elsevier Procedia Computer Science (2019)	Volume 164, 2019, 618-628, ISSN 1877-0509	20

- Reviewer at:
 - Neurocomputing Journal, ISSN: 0925-2312 (2022)
 - International Symposium on Automation, Information and Computing (2021)
 - Yearbook of the Poznan Linguistic Meeting, ISSN: 2449-7525 (2020)
- Professional Organisations:
 - British Computer Society, Chartered Institute for IT - Professional Member (MBCS)
 - Association for Computing Machinery
 - IEEE Computer Society Member
 - Polish Information Processing Society (PTI Member)

Awards

- 2022 Award for the best PhD student at Polish Japanese Academy of Information Technology
- 2022 Rector's Special Award for Research Activities at Kozminski University
- 2021 Best Presentation Award at International Conference on Telemedicine & eHealth for „Liquid State Machine for neuronal computations on High Performance Computing systems”
- 2021 Award for the best PhD student at Polish Japanese Academy of Information Technology
- 2020 Best Presentation Award at International Conference on Telemedicine & eHealth for „Automation of depression screening by applying machine learning to gaze pattern analysis”
- 2017 Aviva Global CIO Award – Global Winner in Outstanding Leadership Category
- 2005 Outstanding Academic Achievement Award for Students from the Polish Minister of Higher Education

Appendix B

Building Neural Simulation Cluster

B.1 Bill of Materials for Neural Simulation Cluster

Building a SBC cluster requires hundreds of hardware components that have to be selected and assembled prior to cluster testing. The items listed below are the complete list of materials that the author selected and collected to build the Neural Simulations Cluster (NSC) for this thesis.

A number of these materials could be supplemented by others (e.g. compatible). A good example of such hardware are the micro USB cables and power supplies, that could be replaced with individual power supplies of each SBC board, or even provided entirely using each board's GPIO expansion header. There is also the option to explore alternatives for a switch / router, SD cards, case, and peripherals. Although this is not a definitive bill of materials, all the key components needed to build the *NSC*, a SBC cluster supporting the preparation of this thesis are listed in the Subsection B.1 below.

The costs of all the materials is the real cost found for the materials purchased since Feb 12th, 2021. By default author made purchases at Kamami.pl, one of the three authorised Raspberry Pi re-sellers in Poland, and one of the most popular on-line electronics stores in Poland. The design of the cluster had to change in early 2022, due to COVID-19 pandemic causing the breakdown of supply chains. When author ordered more RPi boards for the cluster on Jan 15th, 2022, he was soon informed that “the collection of SBC boards has to be postponed” to the end of February 2022, only to be postponed again to March 18th, 2022 and cancelled due to “the lack of availability of the products”.

As a result, the author of this thesis had to change the approach to build the *NSC* from using Raspberry Pi (RPi) boards presented in Table B.1 to ROCKPro64 (RPr), presented in Table B.2.

Table B.1 Hardware specification of Raspberry Pi 4 Model B.

Component	Description
CPU	4 x Cortex-A72 (ARM v8) Broadcom BCM2711 64-bit SoC @ 1.5GHz
RAM	8GB LPDDR4-3200 SDRAM
Wireless Network	2.4 GHz and 5.0 GHz IEEE 802.11ac wireless, Bluetooth 5.0, BLE
Wired Network	Gigabit Ethernet
Ports	2 USB 3.0 ports; 2 USB 2.0 ports Raspberry Pi standard 40 pin GPIO header 2 x micro-HDMI ports (up to 4kp60 supported) 2-lane MIPI DSI display port & CSI camera port 4-pole stereo audio and composite video port
Video	H.265 (4kp60 decode), H264 (1080p60 decode, 1080p30 encode)
Graphics	OpenGL ES 3.0
Storage	Micro-SD card slot for OS and data storage
Power	5V DC via USB-C connector & GPIO header (minimum 3A*) Power over Ethernet (PoE) enabled.
Operating temperature	2.5A power supply can be used if USB peripherals consume less than 500 mA 0 – 50 degrees C ambient

Table B.2 Hardware specification of ROCKPro64.

Component	Description
CPU	4 x ARM Cortex A53 cores @ 1.4GHz + 2 x ARM Cortex A72 cores @ 1.8 GHz
RAM	4GB LPDDR4-3200 SDRAM
Wireless Network	N/A (Optional 802.11ac WiFi module with Bluetooth 5.0)
Wired Network	Gigabit Ethernet
Ports	2 x USB 2.0, 1 x USB 3.0, 1 x USB-C with video out ROCKPro64 standard 40 pin GPIO header, CMOS Sensor port PCIe 4x open-ended slot, eDP port, PI-2 bus, Touch Panel port Display Serial Interface port, Real Time Clock port, 3.5mm jack input Stereo MiPi-SCI port for 12MP cameras, IR R/X port, Lithium battery port
Video	H.265 (4kp60 decode), H264 (1080p60 decode, 1080p30 encode)
Graphics	ARM Mali T860 MP4 GPU
Storage	Micro-SD card slot for OS and data storage
Power	5.5mm barrel power (12V 3A/5A) port
Operating temperature	0 – 65 degrees C ambient

Materials Required for Neural Simulations Cluster

- (1x) Wireless USB Keyboard and Mouse (Logitech K295 Silent Touch, Black),
 - **Cost:** PLN 149 (as of Jan 20th, 2022)
- (1x) microHDMI Goobay 1m HDMI Cable, to connect a monitor:
 - **Cost:** PLN 22 (as of Feb 11th, 2021)
- (1x) Raspberry Pi 4 Model B in the hardware configuration described in Table B.1:
 - **Cost:** PLN 389 (as of Feb 11th, 2021)
- (1x) Radiator Type 37032 (Large):
 - **Cost:** PLN 34.9 (May 26th, 2021)
- (1x) microSD card 32 GB class 10 (GOODRAM):
 - **Cost:** PLN 94.1 (as of Feb 11th, 2021)
- (1x) Ethernet cable, 25cm UTP, Cat 6a:
 - **Cost:** PLN 12.9 (Feb 11th, 2021)
- (1x) Micro USB-to-USB cable (Power)
 - **Cost:** PLN 9.76 (Feb 11th, 2021)
- (1x) Official Raspberry Pi USB-C Power Supply, 5V (Spare):
 - **Cost:** PLN 37 (as of Feb 11th, 2021)
- (3x) ROCKPro64 in the hardware configuration described in Table B.2:
 - **Cost:** PLN 499.15, PLN 1497.45 in total (as of Feb 5th, 2022)
- (1x) Pico Cluster’s “High Power Enterprise Starter Kit” including:
 - Transparent acrylic case
 - Nuts and bolts with base cabling
 - 5 port gigabit switch
 - 80mm Fan

- Internal Power
- **Cost:** PLN 981 in total (as of Feb 6th, 2022)
- (4x) Ethernet Adaptor 270 Degree Female to Male Cat5e/Cat6:
 - **Cost:** PLN 14.99, PLN 56.96 in total (as of March 12th, 2022)
- (4x) microSD card 128 GB class 10:
 - **Cost:** PLN 92.08, PLN 368.32 in total (as of March 12th, 2022)
- **Total cost of PLN 3 652.39, with VAT at 23% (PLN 840.05) is PLN 4 492.44.**

B.1.1 Assembling Hardware Elements

Author built the *Neural Simulations Cluster* using the three RockPro64 boards, and a single Raspberry Pi board. The reasons for using two different SBC board types are described in Tables B.1 and B.2. The build process was started with preparing a three board stack, according to the steps listed below. Author also used the best practice provided by PicoCluster LLC [33], and their Pico 3H Starter Kit as a backbone for the NSC cluster.

1. *Preparing and assembling the SBCs using a standoff.* All the boxes with RPr SBCs had been opened. In the first step author mounted all the heat sinks. As part of the PicoCluster Starter Kit, each radiator had already been equipped with a sticky tape allowing it to attach easily to the main microprocessor of each board. Each of the RPr boards required the setup two radiators. After these were applied, the three RPr SBCs were screwed together using the standoffs. As a result, the backbone of the NSC (the main three RPr computational boards) was assembled, what is presented in Fig. B.1.
2. *Mounting Power Distribution Unit (PDU) and power cables.* The PDU in the Starter Kit was a PicoCluster LLC's custom 10 way, 16 A, 9 Ft. It had 10 Outlets, supply voltage (Vac: 125 Vac). Author mounted the PDU to the side plate of the cluster using long (9.53 mm) screws and nylon washers. After that each power cable was connected to both the PDU and to each of the boards. To avoid cluster's cabling setup to be too unorganised, a cable tie was used to bind all of them together. The PDU is shown in Fig. B.2.
3. *Assembling power supply and its wiring.* A natural place to mount the power supply in most computer cases is to their back panels. The same, standard approach was selected for this cluster case. Firstly, the power supply was mounted with screws and

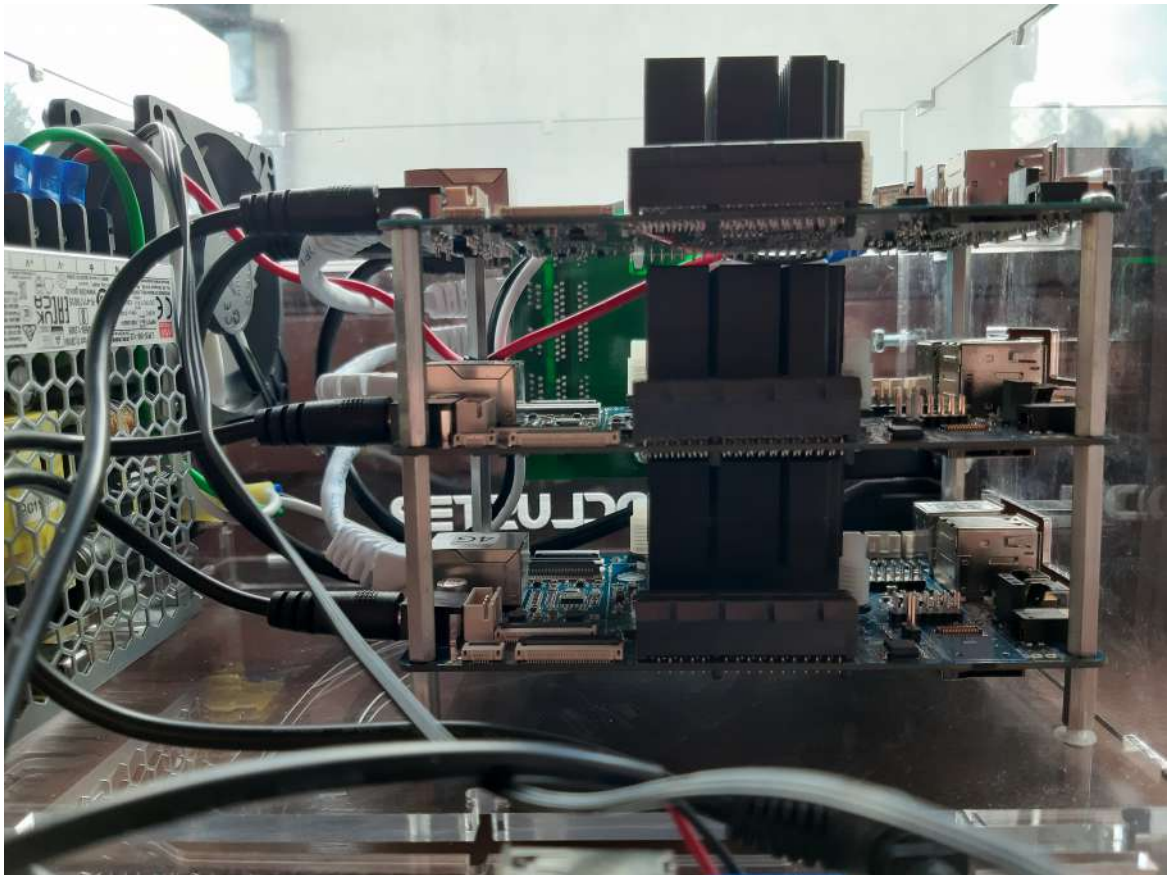


Fig. B.1 Neural Simulation Cluster's three RPr SBCs screwed together using the standoffs.

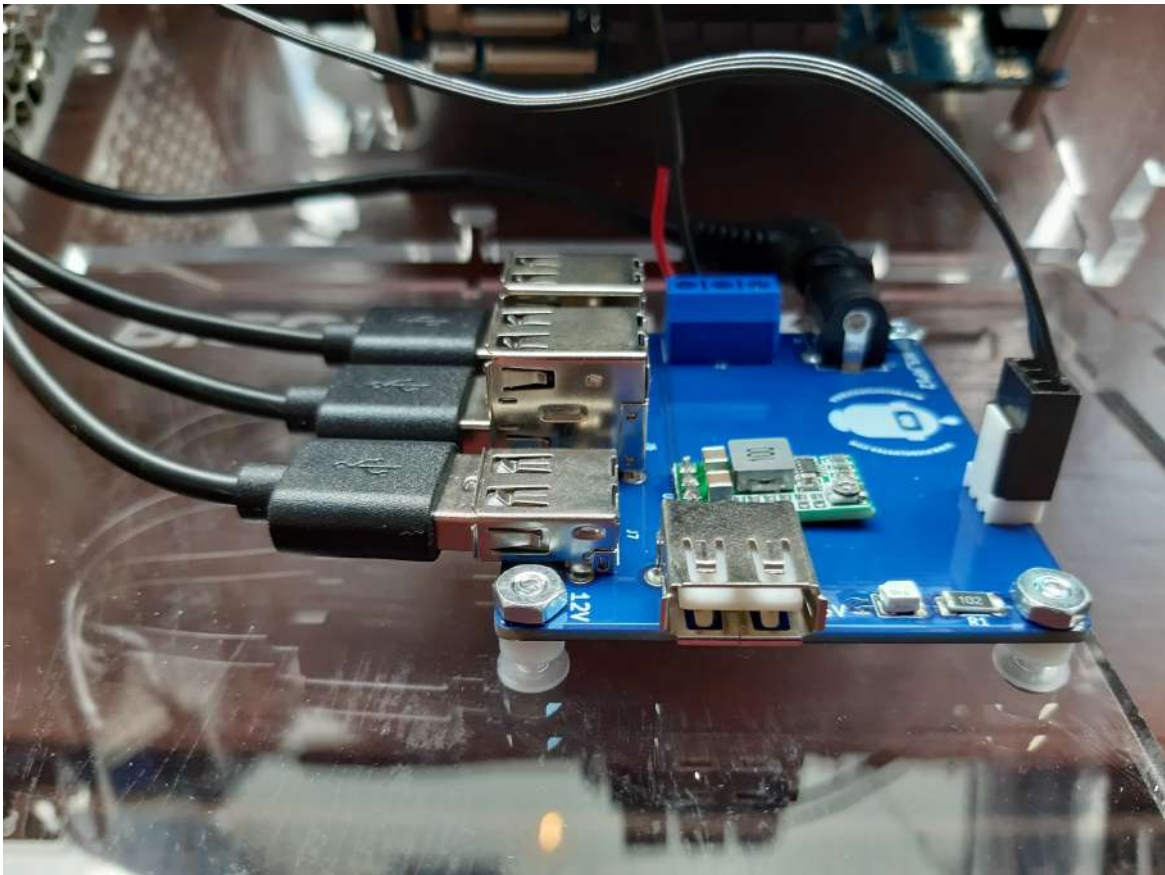


Fig. B.2 Neural Simulation Cluster's power distribution unit and power cables.

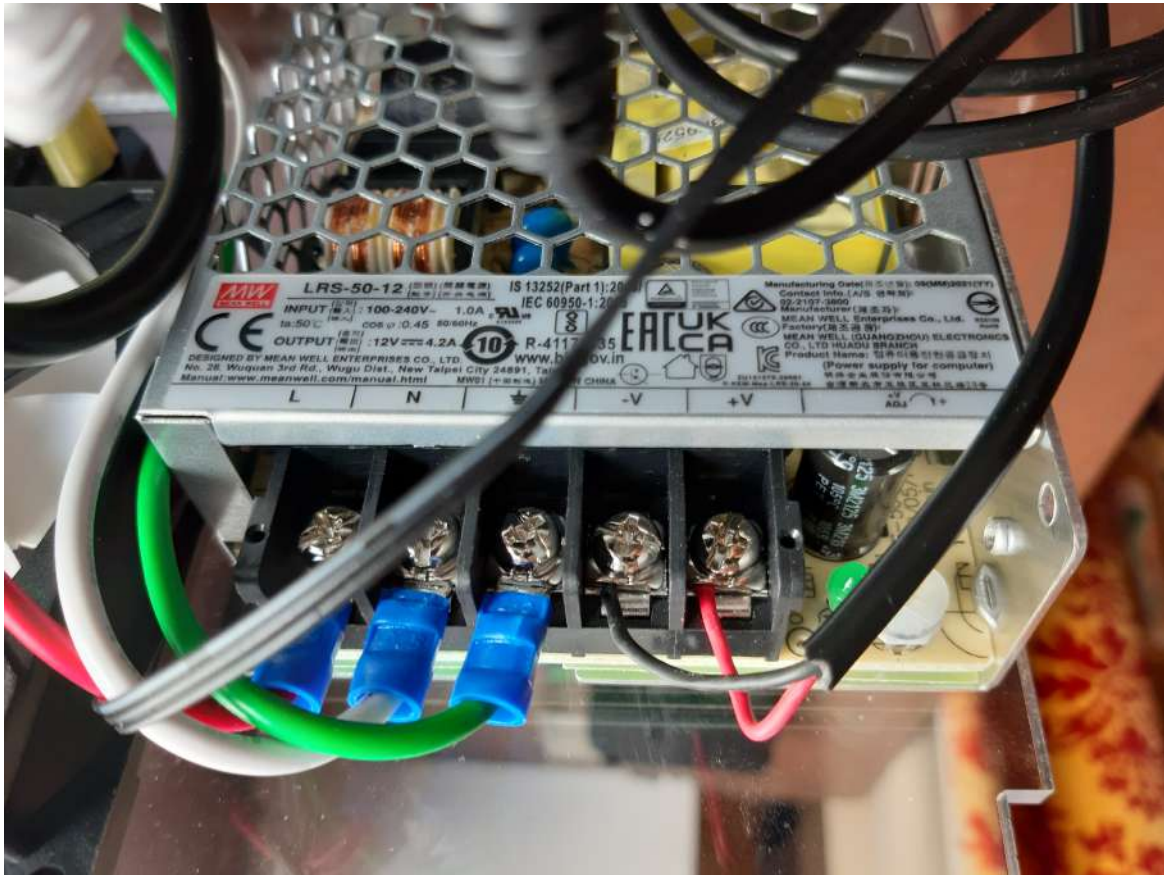


Fig. B.3 Neural Simulation Cluster's A/C power supply and its wiring.

nylon washers to the back plate. Secondly, the A/C power connector was mounted using the 6mm screws and the standard nylon washers. Thirdly, the power input wires were connected to the power supply terminals on the right side. As per manufacturer's designation they are from the right to left: [L] power, [N] neutral, [G] ground. Finally, the pre-wired leads of the board power and fan were connected to the left most terminals. As per the manufacturer's designation, these were from the left to the right: [-V] ground, [+V] positive wires, that are presented in Fig. B.3.

4. *Preparing the NSC's case: the bottom plate, the left side plate with PDU, and the right side plate with the Gigabit Ethernet switch.* The RPr board stack prepared as part of Point 1 was mounted to the base panel using the 9.53 mm screws and standard washers. Firstly, the board standoff was then attached to the upper right side of the base plate in the way that the micro SD card slots can be made available to the outside of the NSC. Secondly, the author mounted the PDU to the side panel using the 12.7 mm screws, 3.175 mm nylon spacers, nuts and washers, as presented in Fig. B.2. Thirdly,

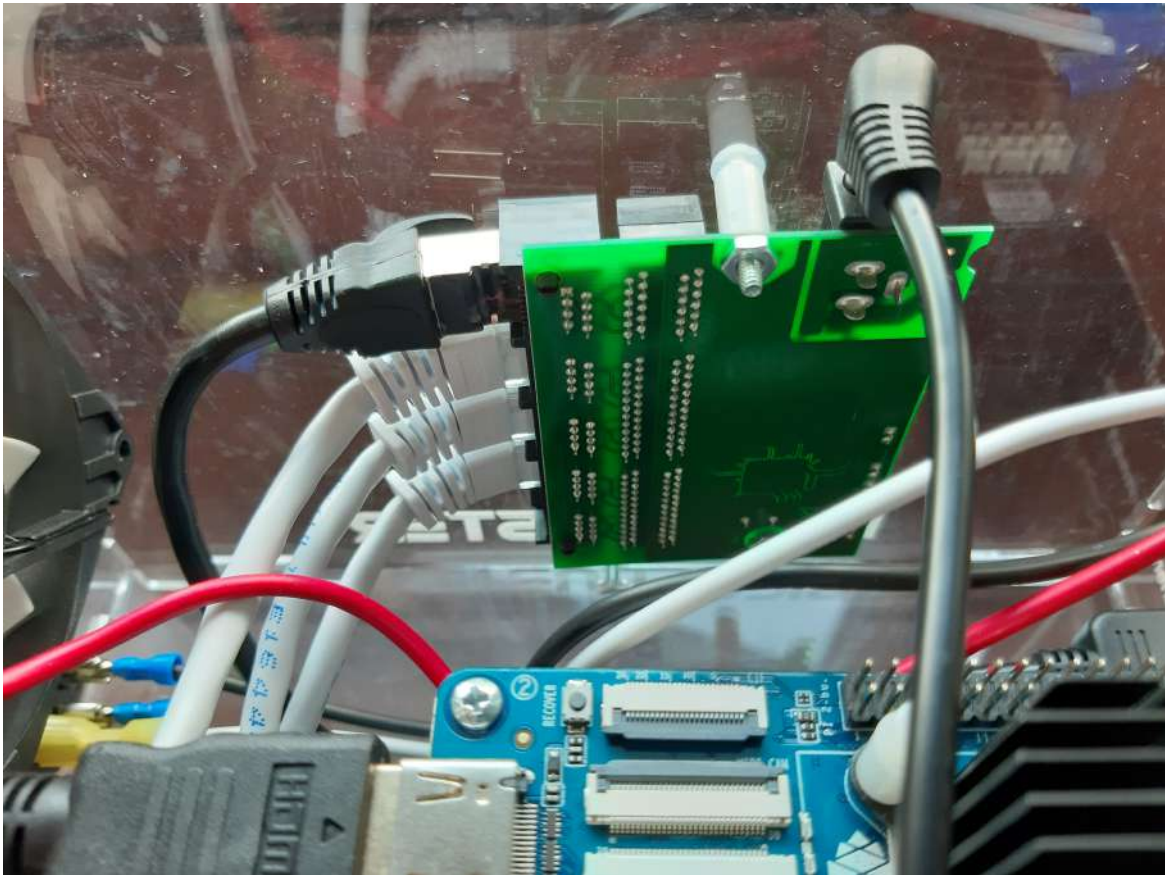


Fig. B.4 Neural Simulation Cluster's Gigabit Ethernet switch providing connectivity between all the NSC nodes, and its cabling.

author mounted the cabling in the way that the right power cable attaches to the right screw terminals, with the black cable plugged into [-], and the red mounted into the right [+5 V], and attached the fan plugs (black wire) to the PDU's fan connectors, as presented in Fig. B.3. Fourthly, the author unboxed the switch package, positioned the switch onto the right plate, and tightened it down with the nuts. As the next step, all the network cables were connected into each RPr board, and to a port in the switch to provide Gigabit Ethernet connectivity between all the NSC nodes. After that process was complete, the switch was mounted to the right side plate, as shown in Fig. B.4.

5. *Organising and finalising the NSC's cabling.* Firstly, author bent and positioned all of the cables along the bottom of the base plate, placing them in the position towards the back of the plate. Secondly, author plugged the black HDMI cable to the HDMI port on the top RPr board. Thirdly, plugged the Ethernet cable to the top port on the switch (black cable), and bent the cable towards the front of the cluster case. Fourthly, author connected the switch power cable into the power port of the switch (black cable).

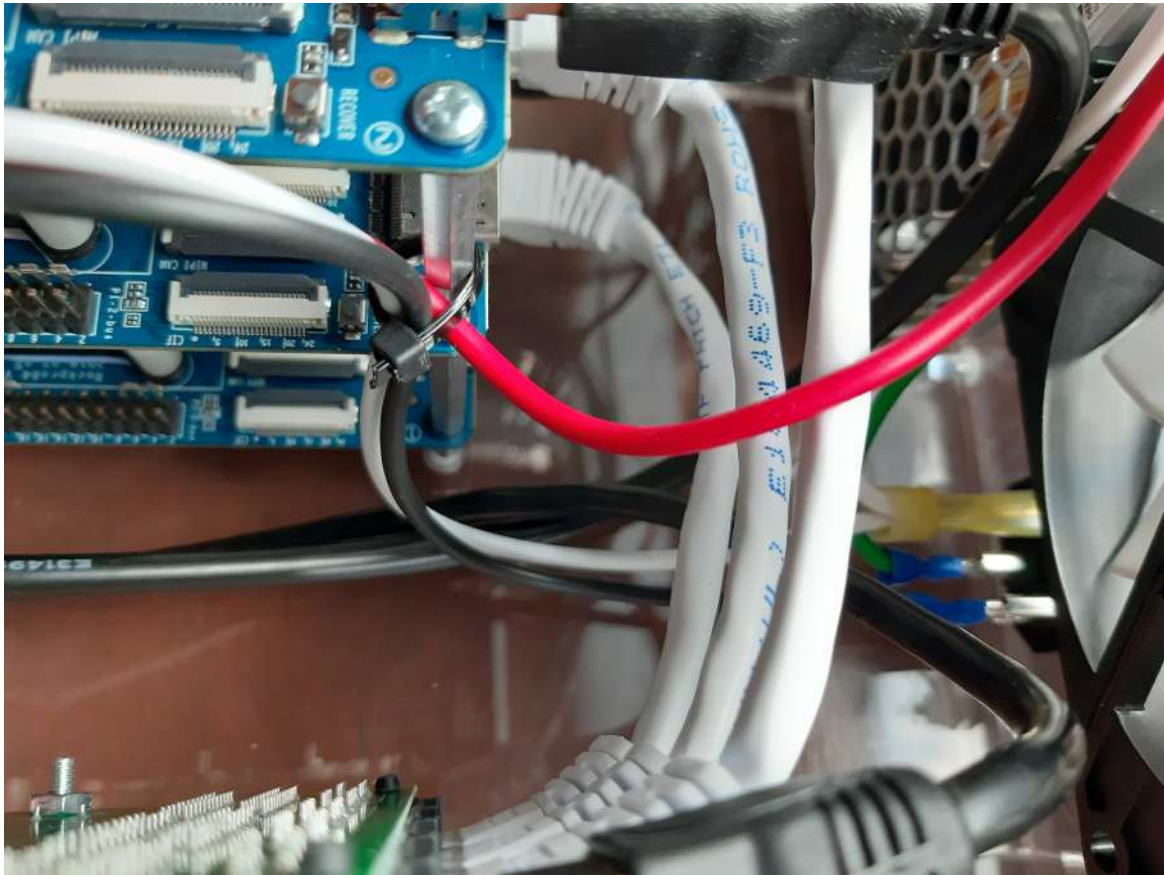


Fig. B.5 Neural Simulation Cluster's internal cabling.

Fifthly, author connected the missing RJ45 Ethernet cable from the bottom slot of the Gigabit switch into the RPi node mounted on the custom top plate (Fig. 5.4). Lastly, author connected the remaining A/C power wires to the On/Off switch on the front plate (red and white wires), and used the cable tie (in the centre) to bind the cabling together, as shown in Fig. B.5.

6. *Finalising the cluster case.* Firstly, author mounted the Ethernet extension and the HDMI cable to the front plate using the 9.53 mm screws and nylon washers. Secondly, he mounted the power switch in the square hole of the front panel. Thirdly, the Gigabit switch plat was assembled to both the base and side plates to form a cube, as shown in Fig. B.6.
7. *Adding the top panel and turning on the NSC system.* The final step is to assemble all the NSC panels together, by attaching the top panel to the base panel, screwing all the plates together, and connecting the power cable (USB-C) and Gigabit network (RJ45)

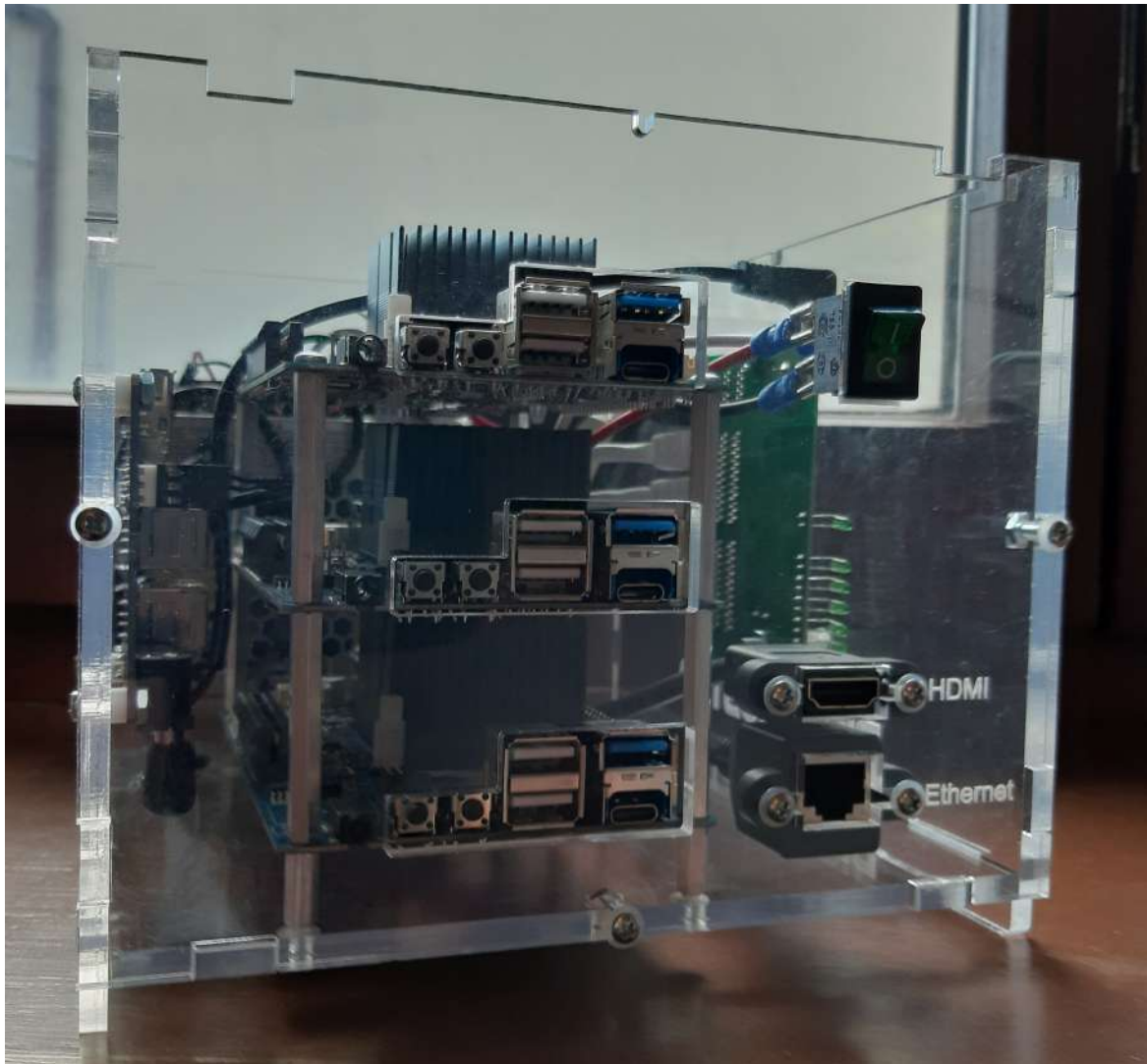


Fig. B.6 A view of Neural Simulation Cluster's assembled front panel.

cable to the top node. After this is completed, the whole NSC can be turned on, and presents itself as in Fig. B.7.

The above instructions have shown how author built the complete *Neural Simulations Cluster*. With all the hardware assembled, the micro SD cards can be inserted into each board and start the cluster to configure the Neural Simulation Pipeline.

B.2 Configuring Neural Simulations Cluster

This subsection presents all the key steps needed to configure the *Neural Simulations Cluster*.

1. The initial step in the NSC configuration was to connect all the cluster nodes to each other. Author configured the nodes in the hosts file. The below listing presents the node layout of:

```

1  10.1.10.240    nsc0
2  10.1.10.241    nsc1
3  10.1.10.242    nsc2
4  10.1.10.243    nsc3

```

This was achieved through generating the SSH key on nsc0 (through *genNscKeys.sh*), and copying the key to all other nodes of the cluster and add them to the *known_hosts* file. The below listing presents this initial configuration that is applicable for each node.

```

1 nspcluster@nsc0:~ $ sh genNscKeys.sh
2 Generating public/private rsa key pair.
3 Created directory '/home/nspcluster/.ssh'.
4 Your identification has been saved in /home/nspcluster/.ssh/
   id_rsa.
5 Your public key has been saved in /home/nspcluster/.ssh/id_rsa.
   pub.
6 The key fingerprint is:
7 a2:06:b8:43:12:ee:28:aa:8d:97:2a:cb:ee:84:c5:55 nspcluster@nsc0
8 The key's randomart image is:
9 +---[RSA 2048]-----+
10 |      E              |
11 |      .              |
12 |.    .              |
13 |.+  .              |
14 |oo+  . S           |
15 |*o  . . .          |

```

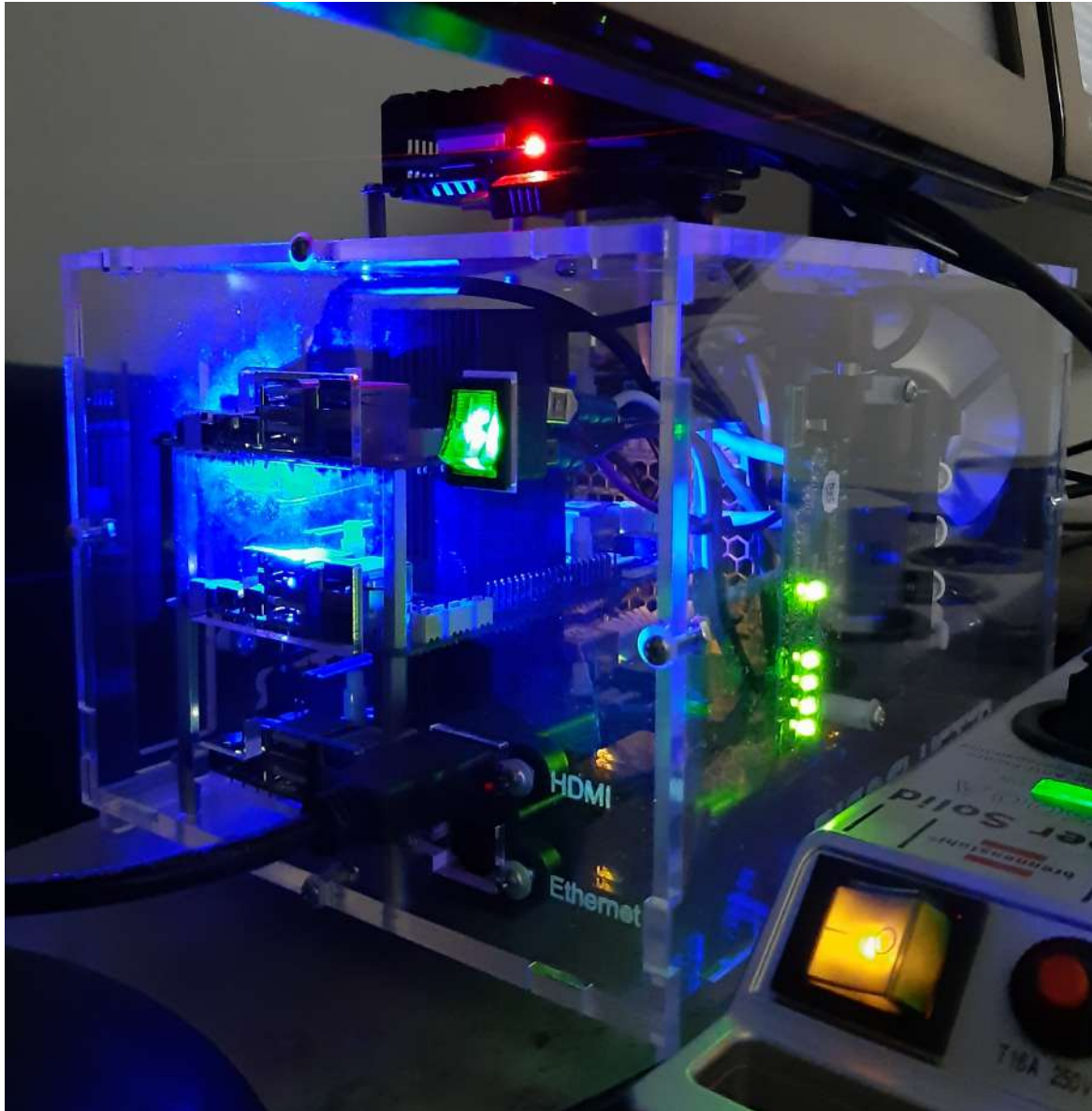


Fig. B.7 Neural Simulation Cluster is up and running with all the four nodes.

```

16 |*o .o          |
17 |*+o.          |
18 |@B.           |
19 +-----+
20 The authenticity of host 'nsc1 (10.1.10.241)' can't be
    established.
21 ECDSA key fingerprint is c2:69:58:03:80:c7:7e:ae:a4:97:c8:ee:4e:
    f0:aa:88.
22 Are you sure you want to continue connecting (yes/no)? yes
23 /usr/bin/ssh-copy-id: INFO: attempting to log in with the new
    key(s), to filter out any that are already installed
24 /usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed --
    if you are prompted now it is to install the new keys
25 nspcluster@nsc1's password:
26
27 Number of key(s) added: 1

```

2. The second step is to log into the machine 2, with: `$sshnscluster@nsc1` and check that the SSH key(s) have been added.

```

1 The authenticity of host 'nsc2 (10.1.10.242)' can't be
    established.
2 ECDSA key fingerprint is c2:69:58:03:80:c7:7e:ae:a4:97:c8:ee:4e:
    f0:aa:88.
3 Are you sure you want to continue connecting (yes/no)? yes
4 /usr/bin/ssh-copy-id: INFO: attempting to log in with the new
    key(s), to filter out any that are already installed
5 /usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed --
    if you are prompted now it is to install the new keys
6 nspcluster@nsc2's password:
7
8 Number of key(s) added: 1

```

3. The same step has to be repeated for machine 3, using: `$sshnscluster@nsc2`

```

1
2 The authenticity of host 'nsc3 (10.1.10.243)' can't be
    established.
3 ECDSA key fingerprint is c2:69:58:03:80:c7:7e:ae:a4:97:c8:ee:4e:
    f0:aa:88.
4 Are you sure you want to continue connecting (yes/no)? yes
5 /usr/bin/ssh-copy-id: INFO: attempting to log in with the new
    key(s), to filter out any that are already installed
6 /usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed --
    if you are prompted now it is to install the new keys

```

```
7 nspcluster@nsc3's password:
```

```
8
```

```
9 Number of key(s) added: 1
```

The same step has to be repeated for machine 4, using: `$sshnscluster@nsc3`

```
1
```

```
2 The authenticity of host 'nsc4 (10.1.10.244)' can't be
   established.
```

```
3 ECDSA key fingerprint is c2:69:58:03:80:c7:7e:ae:a4:97:c8:ee:4e:
   f0:aa:88.
```

```
4 Are you sure you want to continue connecting (yes/no)? yes
```

```
5 /usr/bin/ssh-copy-id: INFO: attempting to log in with the new
   key(s), to filter out any that are already installed
```

```
6 /usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed --
   if you are prompted now it is to install the new keys
```

```
7 nspcluster@nsc4's password:
```

```
8
```

```
9 Number of key(s) added: 1
```

After that we add `nsc0` to its `known_hosts` file, so that it can be accessed by SSH. That is done through connecting to itself and then exiting back to the original shell.

```
1 nspcluster@nsc0:~ $ ssh nsc0
```

```
2 The authenticity of host 'nsc0 (10.1.10.240)' can't be
   established.
```

```
3 ECDSA key fingerprint is c2:69:58:03:80:c7:7e:ae:a4:97:c8:ee:4e:
   f0:aa:88.
```

```
4 Are you sure you want to continue connecting (yes/no)? yes
```

```
5 Warning: Permanently added 'nsc0,10.1.10.240' (ECDSA) to the
   list of known hosts.
```

```
6
```

```
7 The programs included with the Debian GNU/Linux system are free
   software;
```

```
8 the exact distribution terms for each program are described in
   the
```

```
9 individual files in /usr/share/doc/*/copyright.
```

```
10
```

```
11 Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the
   extent
```

```
12 permitted by applicable law.
```

```
13 Last login: Mon Aug 22 15:23:36 2022 from 192.168.1.100
```

```
14 nspcluster@nsc0:~ \ $ exit
```

```
15 logout
```

```
16 Connection to nsc0 closed.
```

4. The final connectivity test ensures that all the NSC nodes can communicate with each other. The script *testNscNodes.sh*, performs a `$df -h` command on all the nodes to show available file systems and mount-points.

```

1 nspcluster@nsc0:~ $ sh testNscNodes.sh
2 testing nsc3
3 Filesystem      Size  Used Avail Use% Mounted on
4 /dev/root       7G   3.3G  3.7G  47% /
5 devtmpfs        459M   0   459M   0% /dev
6 tmpfs           463M   0   463M   0% /dev/shm
7 tmpfs           463M  6.3M  457M   2% /run
8 tmpfs           5.0M  4.0K  5.0M   1% /run/lock
9 tmpfs           463M   0   463M   0% /sys/fs/cgroup
10 /dev/mmcblk0p1  63M   21M   43M  33% /boot
11 tmpfs           93M   0   93M   0% /run/user/1000
12 tmpfs           93M   0   93M   0% /run/user/109
13 tmpfs           93M   0   93M   0% /run/user/1001
14 testing nsc2
15 Filesystem      Size  Used Avail Use% Mounted on
16 /dev/root       7G   3.3G  3.7G  47% /
17 devtmpfs        459M   0   459M   0% /dev
18 tmpfs           463M   0   463M   0% /dev/shm
19 tmpfs           463M  6.3M  457M   2% /run
20 tmpfs           5.0M  4.0K  5.0M   1% /run/lock
21 tmpfs           463M   0   463M   0% /sys/fs/cgroup
22 /dev/mmcblk0p1  63M   21M   43M  33% /boot
23 tmpfs           93M   0   93M   0% /run/user/1000
24 tmpfs           93M   0   93M   0% /run/user/109
25 tmpfs           93M   0   93M   0% /run/user/1001
26 testing nsc1
27 Filesystem      Size  Used Avail Use% Mounted on
28 /dev/root       7G   3.3G  3.7G  47% /
29 devtmpfs        459M   0   459M   0% /dev
30 tmpfs           463M   0   463M   0% /dev/shm
31 tmpfs           463M  6.3M  457M   2% /run
32 tmpfs           5.0M  4.0K  5.0M   1% /run/lock
33 tmpfs           463M   0   463M   0% /sys/fs/cgroup
34 /dev/mmcblk0p1  63M   21M   43M  33% /boot
35 tmpfs           93M   0   93M   0% /run/user/1000
36 tmpfs           93M   0   93M   0% /run/user/109
37 tmpfs           93M   0   93M   0% /run/user/1001
38 testing nsc0
39 Filesystem      Size  Used Avail Use% Mounted on
40 /dev/root       7G   3.4G  3.7G  47% /

```

```

41 devtmpfs          459M      0 459M   0% /dev
42 tmpfs             463M      0 463M   0% /dev/shm
43 tmpfs             463M   6.4M 457M   2% /run
44 tmpfs             5.0M   4.0K 5.0M   1% /run/lock
45 tmpfs             463M      0 463M   0% /sys/fs/cgroup
46 /dev/mmcblk0p1    63M    21M  43M  33% /boot
47 tmpfs             93M      0  93M   0% /run/user/1000
48 tmpfs             93M      0  93M   0% /run/user/109
49 tmpfs             93M      0  93M   0% /run/user/1001

```

5. We can adjust the size of root file system of the NSC nodes to reflect a proper size of their SD cards. This can be done using the *resizeNscFs.sh* script.

```

1 nspcluster@nsc0:~$ ssh nsc2
2 Welcome to Ubuntu 18.04.6 LTS (GNU/Linux 4.4.132-1075-rockchip-
   ayufan-ga83beded8524 aarch64)
3
4 * Documentation:  https://help.ubuntu.com/
5
6 334 packages can be updated.
7 130 updates are security updates.
8
9 Last login: Thu Aug 25 16:29:09 2022 from 10.1.10.240
10 nspcluster@nsc2:~$ sudo su -
11 root@\nsc2:~# sh resizeNscFs.sh
12 Found the start point of mmcblk0p2: 143360
13 Re-reading the partition table failed.: Device or resource busy
14 Ok, Partition resized, please reboot now
15 Once the reboot is completed please run this script again
16 root@\nsc2:~# init 6
17 Connection to nsc2 closed by remote host.
18 Connection to nsc2 closed.
19 nspcluster@nsc0:~$
20 Once all nodes come back, the script must be run a second time.
   ssh to nsc2, sudo to root, then run the resizeNscFs.sh script
   . Once this is completed, then the /dev/root will be expanded
   to full size. Exit twice to return back to nsc0. It will
   look like this. Repeat for all 4 nodes.
21
22 nspcluster@nsc0:~$ ssh nsc2
23 Welcome to Ubuntu 18.04.6 LTS (GNU/Linux 4.4.132-1075-rockchip-
   ayufan-ga83beded8524 aarch64)
24
25 * Documentation:  https://help.ubuntu.com/

```



```
26
27 334 packages can be updated.
28 130 updates are security updates.
29
30 Last login: Tue Aug 23 18:34:27 2022 from 10.1.10.240
31 nspcluster@nsc2:~$ sudo su -
32 root@nsc2:~# sh resizeNscFs.sh
33 Activating the new size
34 resize2fs 1.44.1 (24-Mar-2018)
35 Done!
36 Enjoy your new space!
37 root@nsc2:~# exit
38 logout
39 nspcluster@nsc2:~$ exit
40 logout
41 Connection to nsc2 closed.
42 nspcluster@nsc0:~$
```

After executing the above steps the Neural Simulation Cluster is configured.

B.2.1 Cluster Configuration Steps

Primary Node

- Setting up the configuration on the first NSC node
- Setting up static IP address
- SSH between NSC nodes
- Creating a common user for all NSC nodes
- Generating SSH keys for the common user
- Creating and mounting drives
- Backup NSC (image of each SD card)

Subsequent Nodes

- Potential acceleration of long running simulations, or prototyping distributed models.
- Automatically mounting file-systems on boot-up
- All nodes use the same SD Card image, that can be resized.

Appendix C

Additional Information on Neuronal Models

Table C.1 The standard RetNet simulation parameters for HHLSMs organised into five groups.

Parameter Type	Parameter Values
Main resistances	$R_x = 0.3 \Omega, R_n = 0.33 \Omega$
Capacitance	$C_n = 0.01 \text{ F}$
Potential	$E_n = 0.07 \text{ V}, E_k = 0.0594 \text{ V}$ (soma compartment), $E_k = 0.07 \text{ V}$ (dendrite)
Conductance	$G_K = 360 \Omega^{-1}$ and $G_{Na} = 1200 \Omega^{-1}$ (for each of the ionic channels)
Physical characteristics	A soma with circular shape, diameter of 30μ , and with dendrites and axon length of 100μ

Listing C.1 A code snippet presenting two GENESIS functions written to support the RetNet model, the visual system described in this PhD thesis.

```

1 function make_circuit_2d_output(net, nx, ny, filename)
2   int i,j
3   create spikehistory {net}-history
4   setfield ^ filename {filename}.spike append 0 ident_toggle 1
5   for (i=1; i<={nx}; i={i+1})
6     for (j=1; j<={ny}; j={j+1})
7
8         addmsg {net}_{i}_{j}/soma/spike {net}-history SPIKESAVE
9
10        end
11    end
12
13    echo {net} spike activity saved to file {filename}.spike
14    end
15
16    function make_circuit_3d(protocell, net, nx, ny, nz)
17    str protocell
18    int i,j,k
19
20    for (i=1; i<={nx};i={i+1})
21      for (j=1; j<={ny};j={j+1})
22        for (k=1; k<={nz};k={k+1})
23
24          copy {protocell} {net}_{i}_{j}_{k}
25
26          position {net}_{i}_{j}_{k} { {array_minx} + ({sep_x} * {i}) } \
27                                     { {array_miny} + ({sep_y} * {j}) } \
28                                     { {array_minz} + ({sep_z} * {k}) }
29        end
30      end
31    end
32  end

```

```
// connecting impulses to stimulate retina
make_synapse /input /retina_net_1_1/dend/Ex_channel 2 0
make_synapse /input /retina_net_1_2/dend/Ex_channel 2 0
make_synapse /input /retina_net_1_3/dend/Ex_channel 2 0
make_synapse /input /retina_net_1_4/dend/Ex_channel 2 0
make_synapse /input /retina_net_1_5/dend/Ex_channel 2 0
make_synapse /input /retina_net_1_6/dend/Ex_channel 2 0
make_synapse /input /retina_net_1_7/dend/Ex_channel 2 0
make_synapse /input /retina_net_1_8/dend/Ex_channel 2 0
make_synapse /input /retina_net_2_8/dend/Ex_channel 2 0
make_synapse /input /retina_net_3_8/dend/Ex_channel 2 0
make_synapse /input /retina_net_4_8/dend/Ex_channel 2 0
make_synapse /input /retina_net_5_8/dend/Ex_channel 2 0
make_synapse /input /retina_net_5_7/dend/Ex_channel 2 0
make_synapse /input /retina_net_5_6/dend/Ex_channel 2 0
make_synapse /input /retina_net_5_5/dend/Ex_channel 2 0
make_synapse /input /retina_net_5_4/dend/Ex_channel 2 0
make_synapse /input /retina_net_5_3/dend/Ex_channel 2 0
make_synapse /input /retina_net_5_2/dend/Ex_channel 2 0
make_synapse /input /retina_net_5_1/dend/Ex_channel 2 0
make_synapse /input /retina_net_4_1/dend/Ex_channel 2 0
make_synapse /input /retina_net_3_1/dend/Ex_channel 2 0
make_synapse /input /retina_net_2_1/dend/Ex_channel 2 0
```

	1	2	3	4	5
1	0	0	0	0	0
2	0				
3	0				
4	0				
5	0				
6	0				
7	0				
8	0	0	0	0	0

1_1
1_2
1_3
1_4
1_5
1_6
1_7
1_8
2_8
3_8
4_8
5_8
5_7
5_6
5_5
5_4
5_3
5_2
5_1
4_1
3_1
2_1

	1	2	3	4	5
1				0	4_1
2			0	0	3_2
3		0		0	4_2
4	0			0	4_3
5				0	2_3
6				0	4_4
7				0	1_4
8		0			4_5
					4_6
					4_7
					4_8

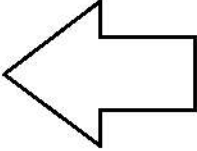


Fig. C.1 A GENESIS code snippet explaining the implementation of synaptic stimulus (patterns of "0" and "1") on the retina's 8x5 neural grid.

Appendix D

Neural Simulation Pipeline

D.1 Repositories

- <https://github.com/KarolChlasta/nsp-code.git>
- <https://github.com/KarolChlasta/nsp-model.git>
- <https://github.com/KarolChlasta/nsp-cluster.git>

D.2 Usage

D.2.1 Neural Simulation Pipeline Installation Steps

1. Install Docker engine:

(a) Install Docker engine from <https://docs.docker.com/engine/install/>.

(b) Log into your local Docker registry.

```
1 $ docker login userAccount
```

(c) Validate your Docker setup by downloading NSP docker image from DockerHub.

```
1 # Linux
2 ./pullNspContainer.sh
```

2. Setup public cloud (AWS):

(a) Create a user account in AWS to access the data stored by NSP.

- (b) Create a technical user account with limited privileges, read/write access to Amazon S3 only, that will be used by NSP scripts to push and pull simulation data to a single location).
 - (c) Install AWS CLI from <https://aws.amazon.com/cli/>.
 - (d) Setup AWS CLI for the accounts created in steps (a) and (b).
3. Download *nsp-code* from GitHub:
- (a) Clone *nsp-code* repository containing pipeline's source code.
 - (b) Clone *nsp-model* repository containing source code of simulations.
4. Setup NSP via environment config file *nsp-code/config.nsp*

```

1 nsp_aws_access_key_id=YOUR_ACCESS_KEY
2 nsp_aws_secret_access_key=YOUR_SECRET_KEY
3 nsp_region=eu-west-1
4 nsp_debug=0
5 nsp_scientist_name=Karol
6 nsp_scientist_surname=Chlasta
7 nsp_scientist_email=chlasta@pja.edu.pl
8 nsp_project_info="Exploring Spiking Neural Networks for Real-
   Time Information Processing"

```

5. Starting NSP container with a simulation engine:

```

1 # Linux
2 ./startNspContainer.sh &

```

Neural Simulation Pipeline Post-installation Steps

1. Select model for your simulation.
2. Execute your simulation with NSP.
 - (a) Get the name of a working NSP container:

```

1 ./showNspContainers.sh
2
3 CONTAINER ID IMAGE NAMES
4 fde8447cad99 karolchlasta/genesis-sim:prod nsp_genesis

```

- (b) Check if the NSP container is configured by fetching the NSP model names already loaded into S3 storage:


```
1 ./listModels.sh
2
3 Container: 7d839b9be3c9
4 2neurons
5 RetNet40
6 RetNet784
7 liquid
```

- (c) Execute a sample simulations on RetNet(8x5,1,25) for “0”, “A”, “1” patterns to check if installation and setup is complete:

```
1 # Linux
2 ./runSampleSim.sh
3
4 # Windows
5 ./runSampleSim.ps1
```

- (d) Note that NSP will print your selected simulation parameters after execution:

```
1 Parameters:
2 1. simulator = genesis
3 2. modelName = RetNet40
4 3. simSuffix = Windows11-8GB
5 4. simDesc = RetNet40
6 5. simTimeStepInSec = 0.00005
7 6. simTime = 1
8 7. columnDepth = 13
9 8. synapticProbability = 0.1
10 9. retX = 5
11 10. retY = 8
12 11. parallelMode = 0
13 12. numNodes = 1
14 13. modelInput = A (12)
```

D.2.2 Neural Simulation Pipeline Simulation Execution Steps

1. NSP facilitates the validation of simulation’s input parameters. Four types of checks on the input parameters are implemented. These are the validation of:
 - (a) a positive integer,
 - (b) an inside of a numerical range,
 - (c) a greater than value,
 - (d) a less than value.

2. NSP saves all the simulation input parameters into the results repository. This is to facilitate the registration of inputs/outputs, and improve the overall traceability of results.

3. To execute simulation, get the name of your working NSP container:

```
1 $ docker ps
2 CONTAINER ID      IMAGE                                     NAMES
3 fde8447cad99     karolchlasta/genesis-sim:prod          nsp_genesis
```

4. Select your simulation engine. At present, NSP allows for choosing between GENESIS and PGENESIS simulation engines (note *parallelMode* = 1 for parallel execution).

5. Execute your simulation, using a set of inputs and parameters (with *runSim.sh* and *runSimulationsManager.sh*; note the concept of NSP variables).

6. Monitor your simulation. A scientist can get information about the running simulation from a few sources, namely by:

- (a) Analysing partial simulation output files in the NSP's central cloud data store, that is accessible via a web browser (AWS Management Console), or AWS CLI.
- (b) Directly monitoring the machine running the NSP container.
- (c) Directly monitoring the NSP container.

i. Enter the NSP container using its interactive mode, to get access to the command prompt in a running container:

```
1 # Linux
2 ./loginNspContainer.sh
```

ii. Check the progress of simulation using a Linux/Unix *tail* command (displaying the last few, in this case 10, lines of simulation's .dat files):

```
1 $ cd /usr/local/genesis-2.4/simulationFolder
2 $ tail -n 10 *.dat
3 -rw-r--r-- 1 root root 440 Nov 10 08:13 RetNet40-1-retina
   .dat
4 -rw-r--r-- 1 root root 818 Nov 10 08:14 RetNet40-1-column
   .dat
5
6 RetNet40-1-retina.dat:/retina_net_4_8/soma/spike
   0.004300
7 RetNet40-1-column.dat:/column_net_1_5_8_195/soma/spike
   0.006300
```

7. Save simulation results, statistics and errors (e.g. using *saveStat.sh*) to the NSP's central cloud storage service:

```
1 $ aws s3 copy . s3://nsp-project/simulations/globalStatistics.nsp
2 $ aws s3 copy . s3://nsp-project/simulations/2022-11-06-120002_RetNet40_5x8_75_9_1_HPI-Docker/
```

Running an Official NSP Container with GENESIS

1. Clone *nsp-code*¹ repository.
2. Build Docker production container from the official NSP image, using the 'prod' tag.
3. Run the standard short simulation for testing (e.g. *runUnitTest.sh*).
4. On a successful test result (e.g. using *runUnitTestCheck.sh*), perform the local NSP container image deployment; push the docker NSP container image from the local images repository (with 'prod' tag) to the DockerHub repository².

Running Simulations with NSP from a local queue

1. Start your NSP container:

```
1 # Linux
2 ./startNspContainer.sh &
```

2. Go to the downloaded or cloned *nsp-code* repository to its sub-folder */scripts/sh/*.
3. To execute your simulation locally, prepare the simulation parameters and add the commands to your local queue, by preparing *localSimulationQueue.nsp*. Sample simulation commands for the *localSimulationQueue.nsp* are presented below:

```
1 runSim.sh --simulator genesis --modelName RetNet40 --simSuffix
  First-Local-Run --simDesc RetNet40 --simulationTimeStepInSec
  0.00005 --simulationTime 1 --columnDepth 3 --
  synapticProbability 0.1 --retX 5 --retY 8 --parallelMode 0 --
  numNodes 1 --modelInput 0;
2 runSim.sh --simulator pgenesis --modelName 2neurons --simSuffix
  First-Local-Run --simDesc 2neurons --simulationTimeStepInSec
  0.00005 --simulationTime 1 --columnDepth 3 --
  synapticProbability 0.1 --retX 5 --retY 8 --parallelMode 1 --
  numNodes 3 --modelInput A;
```

¹Official NSP Code Repository <https://github.com/KarolChlasta/nsp-code.git>

²Official DockerHub NSP Image <https://hub.docker.com/r/karolchlasta/genesis-sim/tags>

4. Run the script *runSimLocally.sh* to start the process of running your simulations from the local queue (one after the other). This functionality is especially useful for long running simulations.

```
1 # Linux
2 ./runSimLocally.sh -f localSimulationQueue.nsp
```

Preparing for Cybernetic Model Development with NSP

1. Create a user account at DockerHub³, to be able to use published Docker images.
2. Install Docker Engine from Docker website⁴.
3. Log into your local Docker registry.
4. Create user account in the public cloud (AWS) for accessing results through NSP (S3) bucket.
5. Create a technical user account with limited privileges (of read/write access to Amazon S3 only), that will be used by NSP to push and pull simulation data to/from Amazon S3 storage service.
6. Install AWS CLI.
7. Setup AWS CLI.
8. Clone *nsp-code* repository with NSP's source code.
9. Clone *nsp-model* repository with cybernetic models' source code.
10. Setup config file for the environment.
11. Run the NSP container.

Cybernetic Model Development with NSP

1. Design a cybernetic model by preparing its source code.
2. Add NSP model variables (e.g. `$modelName$` or `$simulationTime$`). Sample code with the NSP model variables can be found in the code listing below.

³DockerHub <https://hub.docker.com/>

⁴Install Docker from <https://docs.docker.com/engine/install/>

```

1 ...
2 float dt = $simulationTimeStepInSec$ // simulation time step in
   sec
3 ...
4 make_circuit_3d /cell /column_net_1 $retX$ $retY$ $columnDepth$
5 make_circuit_3d_output /column_net_1 $retX$ $retY$ $columnDepth$
   $modelName$-$modelInput$-column
6 ...
7 elif ( $modelInput$ == 1 )
8   make_synapse /input /retina_net_1_1/dend/Ex_channel 2 0
9   make_synapse /input /retina_net_2_1/dend/Ex_channel 2 0
10  make_synapse /input /retina_net_3_1/dend/Ex_channel 2 0
11  ...
12  echo Pattern 1
13  ...
14  // start simulation
15  step $simulationTime$ -time
16  ...

```

3. Add the new model to code repository (e.g. *nsp-model*).
4. Clone the *nsp-model* to a development workstation (Windows or Linux) in order to load the models from the repository (that provides versioning), to the NSP's Amazon S3 (that provides storage for the pipeline).

```

1 $ git clone https://github.com/KarolChlasta/nsp-model.git

```

5. Load the model(s) from your local repository to the Amazon S3 (*loadModels.sh*).

```

1 $ cd nsp-model
2 $ ./loadModels.sh

```

6. Run the simulation by selecting the model (through its name; in the “/model” sub-folder) and other relevant parameters, through the process described for *runSim.sh*.
7. Check the simulation's output (note “/results” sub-folder below).

```

1 # Linux
2 ./listSim.sh
3
4     PRE 2022-11-08-083003_RetNet40_5x8_50_T_1_HPI-Docker/
5     PRE 2022-11-08-091505_RetNet40_5x8_50_K_1_HPI-Docker/
6     PRE 2022-11-08-164749_RetNet40_5x8_50_J_1_HPI-Docker/
7     PRE 2022-11-08-190216_RetNet40_5x8_3_A_1_HPI-Docker/

```

```

8 PRE 2022-11-08-190931_RetNet40_5x8_500_0_1_HPI-Docker/
9 PRE 2022-11-09-150043_RetNet40_5x8_13_0_1_HPI-Docker/
10 PRE 2022-11-09-151850_RetNet40_5x8_13_0_1_Windows11-8GB/
11 PRE 2022-11-09-215406_RetNet40_5x8_200_0_1_HPI-Docker/
12 PRE 2022-11-10-075753_RetNet40_5x8_200_1_1_HPI-Docker/
13 globalStatistics.nsp

```

```

1 $ aws s3 ls s3://nsp-project/simulations/2022-11-09-150043\
  _RetNet40_5x8_13_0_1_HPI-Docker/model/
2
3 2022-11-09 16:00:57      5290 RetNet40-old.g
4 2022-11-09 16:00:59     16788 RetNet40.g
5 2022-11-09 16:00:56     17263 RetNet40.g.org
6 2022-11-09 16:00:55      1088 cell.p
7 2022-11-09 16:00:58     14443 simulationInfo.out
8 2022-11-09 16:00:55      1961 functions.g
9 2022-11-09 16:00:54     13575 patterns.g
10 2022-11-09 16:00:53      1559 protodefs.g
11 2022-11-09 16:00:52      1249 start.sh
12
13 $ aws s3 ls s3://nsp-project/simulations/2022-11-09-150043\
  _RetNet40_5x8_13_0_1_HPI-Docker/results/
14
15 2022-11-09 16:02:10     721644 RetNet40-0-column.dat aut
16 2022-11-09 16:02:11     35200 RetNet40-0-retina.dat
17 2022-11-09 16:02:12        250 RetNet40.err
18 2022-11-09 16:02:13    2283266 RetNet40.out
19 2022-11-09 16:02:15     14510 simulationInfo.out

```

D.2.3 Neural Simulation Pipeline Finalisation and Cleanup Steps

1. Retrieve the simulation results. Below is the command for downloading the simulation files:

```

1 # Linux
2 ./downloadSim.sh --uri s3://nsp-project/simulations
  /2022-11-06-120002_RetNet40_5x8_75_9_1_HPI-Docker/

```

2. Retrieve simulation's statistics only:

```

1 # Linux
2 ./downloadSim.sh --uri s3://nsp-project/simulations/
  globalStatistics.nsp

```

3. Perform the data analysis.
4. Stop and delete the NSP container from your local store:

```
1 # Linux
2 ./stopNspContainer.sh
```

5. Remove the NSP container's image to save your disk space by running the below command:

```
1 ./deleteNspImages.sh
```

6. Perform your data analysis using the results recorded in NSP's central simulation store.

D.3 Scripts

Table D.1 Full list of scripts in Neural Simulations Pipeline and in Neural Simulation Cluster.

User Scripts	Container Scripts	Cluster Scripts
buildNspImage.sh	calculatePeriod.sh	resizeNscFs.sh
deleteNspImages.sh (.ps1)	configAWSCLI.sh	resizeNscNodes.sh
downloadSim.sh (ps1)	downloadModel.sh	restartNscNodes.sh
getAWSCredentials.ps1	listSim.sh	stopNscNodes.sh
listModels.sh (.ps1)	runSim.sh	testNscNodes.sh
listSim.sh (ps1)	runSimLocally.sh	genNscKeys.sh
loadModels.sh (.ps1)	runSimulationsManagerS3.sh	
loginNspContainer.sh (.ps1)	saveStat.sh	
pullNspImage.sh (.ps1)	showStat.sh	
pushNspImage.sh	showSystemInfo.sh	
runSampleSim.sh (.ps1)	validatePositiveInteger.sh	
runSimLocally.sh (.ps1)	validateRange.sh	
runSimRemotely.sh (.ps1)	validateRealNumber.sh	
runUnitTest.sh (.ps1)	writeDebug.sh	
runUnitTestCheck.sh	writeOutput.sh	
showNspContainerLogs.sh (.ps1)		
showNspQueue.sh (.ps1)		
startNspContainer.sh (.ps1)		
stopNspContainer.sh (.ps1)		