Brain-Inspired Placement and Routing for Neuromorphic Processors

Dissertation

zur

Erlangung der naturwissenschaftlichen Doktorwürde

(Dr. sc. UZH ETH Zürich)

vorgelegt der

Mathematisch-naturwissenschaftlichen Fakultät der Universität Zürich

und der

Eidgenössischen Technischen Hochschule Zürich

von

Vanessa Rodrigues Coelho Leite

aus

Brasilien

Promotionskommission

Prof. Dr. Giacomo Indiveri (Vorsitz)

Dr. Matthew Cook
Dr. Yulia Sandarmiskaya
Prof. Dr. Valerio Mante
Prof. Dr. Eleni Vasilaki
Prof. Dr. Catherine Schuman

Zürich, 2023

"We can't afford that all of our research is devoted to the machine, because what we are trying to learn about isn't the machine we are building: it's the brain."

Misha Mahowald

Abstract

The significant energy costs of Deep Neural Network (DNN) and Artificial Intelligence (AI) algorithms are pushing the development of domain-specific hardware accelerators. Neuromorphic processors are a class of AI hardware accelerators that implement computational models of Spiking Neural Networks (SNNs) adopting in-memory computing strategies and brain-inspired principles of computation. However, the requirement of SNN hardware accelerators to store the state of each neuron, combined with their in-memory computing circuit design techniques, leads to substantial area consumption figures, which limits the sizes and numbers of parameters of the networks that they can implement.

The current strategy used to support the integration of large SNN models in these accelerators is to use multi-core architectures. In these architectures, each core either *emulates* with analog circuits or *simulates* with time-multiplexed digital circuits neuro-synaptic arrays. The synaptic weight matrix and the connectivity routing memory blocks occupy a significant proportion of the total layout area.

Finding trade-offs to optimize weight-matrix, connectivity, and routing memory structures in multi-core neuromorphic processors can significantly impact their total chip die area and the size of the networks they can implement. Following the original neuromorphic engineering approach, we look at animal brains for inspiration and propose a brain-inspired routing and placement strategy to reduce memory in multi-core neuromorphic hardware.

In animal brains, computation, and other functions emerge from the interaction of neural areas. Brain networks express modular, *small-world*, heavy-tailed characteristics. In small-world networks, most edges form small, densely connected clusters, while others maintain connections between these clusters. By restricting the neuromorphic processor to implement small-world SNNs, we can dramatically reduce the memory required to specify the routing and connectivity schemes while still supporting a wide range of computations for solving pattern recognition and signal processing tasks.

Specifically, we show that, by focusing on small-world network connectivity, we can implement trade-offs that minimize memory consumption requirements while still enabling the design of SNN architectures that can solve a wide range of relevant "edge-computing" problems, i.e., the types of sensory-motor processing problems that animals must solve in the real world.

Placing an SNN onto neuromorphic hardware is a mandatory step needed to exploit the advantages of the hardware, and each type of hardware has its own set of tools to make it appealing to SNN developers, which are often not familiar with the hardware complexities. In this Thesis, we use a hardware-software co-design approach, where the memory minimization strategies validated in software lead to routing circuit specifications for new neuromorphic chip designs.

The contributions of this Thesis are two-fold: i) a novel routing architecture designed to support small-world networks, and ii) a new placement algorithm to provide specifications for new hardware designs. When developing the placement algorithm, we consider requirements derived from hardware design choices proposed by the chip designers, which define constraints on the algorithm. In this way, we optimize software and hardware together.

Our co-design approach reduces the memory necessary to place and route networks that follow a small-world structure while not limiting the possible applications. Additionally, our placement algorithm can find optimal solutions for networks that follow our canonical design and can place deviations from them without diverging too much from the ideal case. The simultaneous design of a place and route scheme allows us to design a new multi-core SNN chip to handle more extensive networks with a minimum of memory consumption.

Acknowledgments

During the last four years, every time I was asked, "what do you do?" I proudly would say, "I am a Ph.D. student in Neuroinformatics", and I knew that shortly after, I would say, "we try to understand how the brain works and mimic its behavior in a new machine called neuromorphic hardware". People often would be surprised and impressed: "you must be smart!".
I have never considered myself smart. I have spent a lot of effort and energy to build who I am today and to acquire the knowledge I have. I consider myself hardworking, privileged, and amazingly lucky.
It was not because I am smart, nor only because of my efforts, that this Thesis is now complete. This Thesis is a product of many people that crossed my life.

I will start by saying how grateful I am for the support of my whole family, that even though they didn't really understand, they accepted my efforts in moving out from my hometown to do a master's and, later on, to cross the Atlantic for the Ph.D. I am incredibly grateful to my mom, Andrea, who, in her unique way, always pushed me to study and do better; and for my sis, Brenda, that held things together back home and gave me the love of my life, Benicio: one more reason to keep fighting and pursuing my dreams.

It would be naive to believe this journey started when Giacomo accepted me into the ZNZ program. In fact, it started when my bachelor's Professors told me I could do more and gave me support to do so. I am so grateful to all my former professors, especially Anselmo Paiva, Aristófanes Silva, Carlos de Salles, and Marcelo Gattass. They were keystones for my academic career. I cannot express how much I value their opinions and all the help I got throughout my Bachelor, Master, and even Ph.D.

I need to thank all my Brazilian friends, either in Zurich or back in Brazil, who helped me feel that home was not that far. Thank you so much, Friedrich Garcez, Isadora Martins, Camila Silva, Humberto Victor, Adriano Reis, Taniel Martins, Bethania Souza, Mariana Gliesh, Fabiola Maffra (and Lucas e Mia), Eliana Goldner, Paula Rodrigues, Eduarda Morsch, Priscilla Matos, and Lais Guimaraes for making my days brighter in this cloudy, cold and grey city.

Working between two groups at the Institute of Neuroinformatics (INI) has been tremendous, and INI has become a second home for me. Thank you to everyone at INI, everyone who gave me insightful feedback after lab meeting talks, Journal Clubs, the admin team, and all my colleagues from the Neuromorphic Cognitive Systems and the Cookies, in particular to Julia Buhmann, Nils Eckstein, Moritz Milde, Ethan Palmiere, Xander Nedergaard, Melika Pavyland, Mohammad Ali, Matteo Cartiglia, Alpha Renner, and Carsten Nielsen. Thank you to Zhe Su, Junren Chen, and Adrian Whatley for the close discussions and all the help with this Thesis.
Many, many thanks in particular to the people that welcome me in their private lives: Karlita Burelo, Nicoletta Risi, Raphaela Kreiser, Renate Krause, Giorgia Dellaferrera, Arianna Rubino, Matilde Tristany, Dmitrii Zendrikov, Lucas Pompe, Gala Sanchez, Tavo Siller, Luca Zuccarini (and Braska!), Marco Eppenberger, and Andrea Magazzini. I could not be more grateful to get to know you all, and I genuinely appreciate your friendship and all the hugs and love.
Thank you to all the people involved, directly or indirectly, for providing me with feedback and support. Thank you to Vanessa Machado, Roberto Azevedo, and the EU grant.

I have no words to say thanks to Matthew Cook. Matthew was by my side since my first steps in Switzerland. He cried with me when I faced problems; he helped me to find solutions when

I could not see one anymore. He advised me not only in my projects and this Thesis but also in my personal life and helped me to create a sense of future. Thank you, Matt, for all the beers and gin tonics we had together, for bringing me gin and marshmallows when we elected the worst president in Brazilian history. Thank you for breaking down my stressful moments and making me go away for a bit when I needed to but could not see it. Thank you for always being available when I needed to vent out and complain about nothing and everything. I wish everyone could have a Matthew in their lives.

And, finally, my biggest and deepest gratitude to Giacomo Indiveri. Giacomo has been the best supervisor I never could dream of. I am immensely grateful for the opportunity, the project, all the pieces of advice, and his patience and support. Not only he guided me in this academic career, but he also supported me in fighting for gender equality and opened paths for what is yet to come. It was his support that allowed me to have a voice and to feel empowered and confident. And even with his limited time, he always managed to find space for me, my complaints, and my doubts. Sorry, not sorry, Giacomo, for all the TikToks, and I can only hope that one day I will be as inspiring to others as you are to me.

All-in-all, I cannot pretend I had no fears or to see this time with rose-tinted glasses. However, I cannot deny this incredible feeling of gratitude. This Ph.D. has been quite a journey. It is not only about all the knowledge and academic formation I got. It is, and I dare to say in its biggest part, about all the people I had around throughout these years. You all have been with me through the highs and lows; your support got me through, made me keep going, and brought me here today.

During these years, I have loved and been loved; I have read and written, traveled and made a home; I have thought and dreamed, learned and taught; I have laughed and cried, I fought and quietened down; I have built my own way and accepted known paths; I made friends and changed who I am.

These Ph.D. years have been, in themselves, an enormous privilege and my biggest adventure. I know I have complained. Nevertheless, I will miss every single bit of it.

Agradecimentos

Durante os últimos quatro anos, sempre que alguém me perguntava "o que você faz?", eu cheia de orgulho respondia "sou doutoranda em neuroinformática". E eu sabia que logo depois eu diria "a gente tenta entender como o cérebro funciona pra reproduzir esse comportamento em um tipo especial de computador chamado hardware neuromórfico". As pessoas geralmente ficam surpresas e impressionadas: "nossa, você deve ser muito inteligente!".
Eu nunca me considerei inteligente. Eu me esforcei bastante pra ser quem eu sou hoje, e pra ter todo o conhecimento que tenho. Eu me considero muito esforçada, privilegiada e incrivelmente sortuda.
Não é por que eu sou inteligente, ou exclusivamente por meus esforços que hoje essa tese é completa. Essa tese é resultado do apoio de muitas pessoas que passaram pela minha vida.

Quero começar dizendo como sou grata por todo o apoio de toda a minha família que mesmo sem realmente entender, aceitaram meus esforços em sair de São Luís pra fazer meu mestrado no Rio; e, depois, de cruzar o Atlântico pra fazer o doutorado na Suiça. Sou incrivelmente grata pela minha mãe, Andréa, que com seu jeitinho único, sempre me incentivou a estudar e ser melhor; e pela minha mermã, Brenda, que segurou as pontas dentro de casa e me deu o amor da minha vida, Benicio: uma razão a mais para continuar lutando e correr atrás dos meus sonhos.

Seria ingênuo acreditar que essa jornada se iniciou quando Giacomo me aceitou no programa de doutorado. De fato, essa jornada começou quando meus antigos professores me disseram que eu podia fazer mais, e me deram suporte para tanto. Eu sou muito grata aos meus antigos professores, em especial, Anselmo Paiva, Aristófanes Silva, Carlos de Salles, e Marcelo Gattass. Eles foram peças fundamentais na minha carreira academica. Não posso espressar o quanto eu valorizo as opiniões deles, e toda a ajuda que eles me ofereceram durante meu bacharelado, mestrado, e até agora no doutorado.

Eu preciso agradecer a todos os meus amigos BRs, que em Zurique ou mesmo no Brasil, me ajudaram a sentir que minha zona de conforto não estava tão longe. Muito obrigada, Friedrich Garcez, Isadora Martins, Camila Silva, Humberto Victor, Adriano Reis, Taniel Martins, Bethania Souza, Mariana Gliesh, Fabiola Maffra (e Lucas e Mia), Eliana Goldner, Paula Rodrigues, Eduarda Morsch, Priscilla Matos, e Lais Guimarães por iluminarem meus dias nessa cidade nublada, fria e cinzenta.

Trabalhar em dois grupos no Instituto de Neuroinformática (INI) tem sido maravilhoso, e o INI se tornou minha segunda casa. Muito obrigada a todos do INI, todo mundo que me deu sugestões e comentários sobre meu projeto, e como preparar apresentações, ao time da administração, e a todos os meus colegas, em especial a quem esteve comigo no Grupo de Sistemas Neuromórficos Cognitivos e no grupo do Matthew Cook: Julia Buhmann, Nils Eckstein, Moritz Milde, Ethan Palmiere, Xander Nedergaard, Melika Pavyland, Mohammad Ali, Matteo Cartiglia, Alpha Renner, e Carsten Nielsen. Obrigada Zhe Su, Junren Chen, e Adrian Whatley pelas conversas, ideias e ajuda direta nessa tese.
Muito, muito obrigada em particular para as pessoas que me receberam em suas vidas: Karlita Burelo, Nicoletta Risi, Raphaela Kreiser, Renate Krause, Giorgia Dellaferrera, Arianna Rubino, Matilde Tristany, Dmitrii Zendrikov, Lucas Pompe, Gala Sanchez, Tavo Siller, Luca Zuccarini (and Braska!), Marco Eppenberger, e Andrea Magazzini. Eu não poderia ser mais feliz em ter conhecido todos vocês, e eu genuinamente aprecio a amizade, todos os abraços e amor que recebi.
Obrigada a todas as pessoas que, direta ou indiretamente, me deram suporte: Vanessa Machado,

Roberto Azevedo, e minha bolsa da União Européia.

Eu não tenho palavras pra agradecer Matthew Cook. Matthew esteve ao meu lado desde meus primeiros passos na Suiça. Ele chorou comigo quando eu encontrei problemas; ele me ajudou a encontrar soluções quando eu não conseguia ver saídas. Ele me aconselhou não apenas nos meus projetos e nessa tese, mas também na minha vida pessoal, e me ajudou a criar uma sensação de futuro. Obrigada, Matt, por todas as cervejas e gin tonicas que bebemos, por me trazer gin e marshmallows quando elegemos o pior presidente da história do Brasil (#elenao). Obrigada por aliviar meus momentos de estresse e me distrair quando eu precisava mas não conseguia perceber. Obrigada por sempre estar disponível quando eu precisava colocar pra fora minhas frustrações e reclamar sobre tudo e qualquer coisa. Seria maravilhoso que todo mundo pudesse ter um Matthew em suas vidas.

E, finalmente, minha mais profunda gratidão ao Giacomo Indiveri. Giacomo tem sido o melhor orientador, como eu jamais poderia imaginar. Eu sou imensamente grata pela oportunidade, pelo projeto, por todos os conselhos, pela paciência, e pelo apoio. Não apenas ele me guiou nessa jornada acadêmica, mas também me apoiou quando lutei por igualdade de gênero, e ele abriu caminhos para o que ainda está por vir em minha vida. Foi o apoio dele que me permitiu ter uma voz, e me sentir empoderada e confiante. E mesmo com a agenda completamente cheia, ele sempre encontrou tempo pra mim, pra minhas reclamações e minhas dúvidas. Sinto muito, mas nem tanto, por todos os TikToks, e eu só posso desejar que um dia eu seja inspiração para os outros como você é pra mim.

De modo geral, não posso fingir que não tive medos, nem dizer que o doutorado foi um mar de rosas. Entretanto, eu não posso negar esse imenso sentimento de gratidão. Esse doutorado tem sido uma jornada incrível. E não é apenas sobre todo o conhecimento e formação acadêmica que recebi. É, e eu arrisco dizer na maior parte, sobre todas as pessoas que eu tive ao redor durante esses anos. Todos vocês estiveram comigo nos bons e maus momentos, e foi o apoio de vocês que me ajudou a continuar, e que me trouxe até aqui.

Durante esses anos, eu amei e fui amada; eu li e escrevi, viajei e construí um lar; Eu pensei e sonhei, aprendi e ensinei; Eu ri e chorei, eu lutei e fiquei quieta; Eu andei pelo meu próprio caminho e aceitei rotas que já eram conhecidas; Eu fiz amigos e mudei quem eu sou.

Esse doutorado foi um enorme privilégio e minha maior aventura. Eu sei que eu reclamei. E mesmo assim, eu vou sentir falta de cada pedacinho dele.

Contents

Acronyms

AER Address-Event Representation

AE Address-Event

AI Artificial Intelligence

ANN Artificial Neural Network

API Application Programming Interface

CAM Content Addressable Memory

CNN Convolutional Neural Network

CPU Central Processing Unit

CTXCTL CortexControl

DNN Deep Neural Network

DPI Differential Pair Integrator

DRAM Dynamic Random Access Memory

DYNAP Dynamic Neuromorphic Asynchronous Processor

FNN Feed-forward Neural Network

GT Ground Truth

GUI Graphical User Interface

H&H Hodgkin & Huxley

I&F Integrate-and-Fire

INI	Institute of Neuroinformatics
LI&F	Leak Integrate-and-Fire
ML	Machine Learning
MLP	Multilayer Perceptron
NCS	Neuromorphic Cognitive Systems
NoC	Network-on-Chip
PC	Personal Computer
RAM	Random Access Memory
RNN	Recurrent Neural Networks
SNN	Spiking Neural Network
SOC	System-On-Chip
SRAM	Static Random Access Memory
STDP	Spike-Timing Dependent Plasticity
TCAM	Ternary Content-Addressable Memory
VLSI	Very Large Scale Integration
WTA	Winner-Take-All

*"O-M-G. What am I doing?
shrubba-shrubba."*

Vanessa Leite, *many times during the last four years*

1. On brains and computers

The comparison between our brains and our computers seems, at first, unfair: give a computer a complex equation, and you will get an answer within seconds. And in contrast, a human brain needs significantly more time to perform the same calculation. However, we find in the brain similar structures to what we design in computers: there is a "Central Processing Unit (CPU)", a hard drive and Random Access Memory (RAM), a camera with a "computer vision" engine, a natural language processor, and so on.[1] It is, therefore, compelling to see our brains as equivalent to our computers! Nevertheless, they are different: the brain has evolved to interact with the environment and help its body survive and reproduce, while computers were created to assist mathematical systems and perform calculations and computations (Abbate, 1999; Ceruzzi, 2003). From the first computers, with Turing and von Neumann, to IBM creating the Personal Computers (PCs) (Abbate, 1999), and the popularization of computers and mobile devices, we adopted them in our daily lives. This popularization allowed us to use computers as truly versatile machines instead of hardwired circuitries made for a particular task. Now, computers not only perform calculations and computations, but they are also used to receive data through their own sensors, process it, and respond to the environment using AI (Brown et al., 2020; Kato et al., 2015; Zhang and Tao, 2020). The transformation from calculators to machines interacting with the environment was only possible through a chain of incredible advances: gradient descent (Amari, 1967; Lemaréchal, 2012), backpropagation (Werbos, 1990; Schmidhuber, 2014), unsupervised learning algorithms (Dayan, 1999; Erhan et al., 2010; Gütig, 2016), and the development of brain-inspired networks such as Convolutional Neural Networks (CNNs) (Scherer et al., 2010; Ranzato and LeCun, 2007), and DNNs (LeCun et al., 2015). All of this, and the acknowledgment that the hardware is an essential factor in our capacity to make computers learn and that brains and computers are bounded by physical laws (Schmidhuber, 2022), have led to the development of more complex processing circuits and our first attempts at mimicking the human brain or human intelligence.

In recent years, AI has undergone an extraordinary and astonishing revolution. AI has touched many brain-based cognition and intelligence levels. Nowadays, our computers can beat humans in games (Koch, 2016; Chen et al., 2018) and can do an impressively good job at complex machine vision tasks, such as face recognition or digital art. For instance, DALL-E (Ramesh et al., 2022) can create realistic images and art from a description in natural language, as shown in Fig. 1.1.

However, classical AI algorithms such as DNNs typically use "brute force" approaches to perform their tasks. Furthermore, they require a vast set of training examples and lots of training cycles. Moreover, adding a single new input to the training demands retraining the models from scratch (Zhu and Klabjan, 2021). The (re)training of such large models does not come without a cost. We have seen the cost of training AI models doubling every ~ 4 months, as shown in Fig. 1.2, reaching petaflops/s per day, and the power consumption increased $300,000$-fold from 2012 to 2018, raising real concerns about energy costs for the world (OpenAI, 2018; Numenta, 2022; Labbe, 2021; Economist, 2020).

Our computers and AI evolved because there are more than 7 billion brains on the planet, and a portion of them can share knowledge successfully to make science progress. And we have been working on improving computers and AI for around six decades now.

Besides sharing knowledge, as humans, our brains are marvelous in cognitive and information-processing tasks, such as object recognition or complex scene analysis and understanding. Indeed, our AI models can beat us in games and produce excellent results in computer vision tasks. However, our brains are even more breathtaking: they rely only on a few data, using multiple

[1] The equivalence is to our frontal cortex, hippocampus, eyes and visual cortex, temporal lobe, and more.

Figure 1.1.: By only offering a small text and a style of painting, AI can produce impressive results. This image was generated by inputting "female neuroscientist is mapping a neural network onto a computer chip digital art" in the DALL-E system (Dayma et al., 2021).

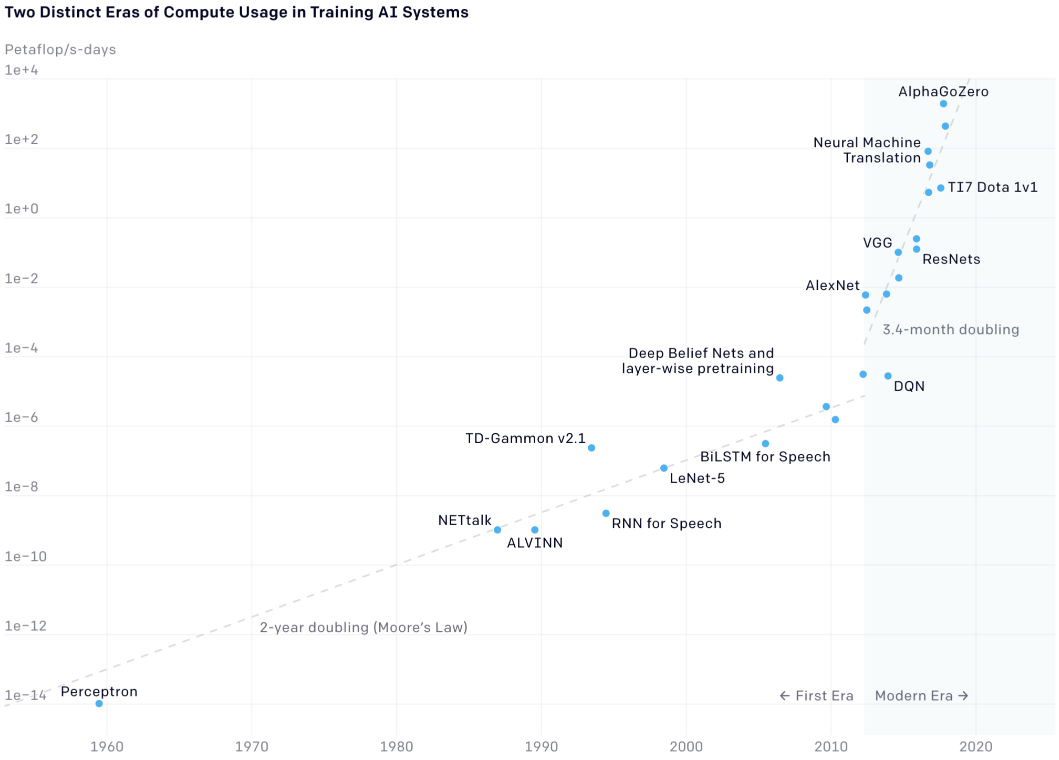


Figure 1.2.: The amount of computational power used in the largest AI training runs has been increasing exponentially, doubling every ∼ 4 months. Since 2012, there has been an increase of more than $300,000$x in power consumption. Figure from (OpenAI, 2018).

modalities combined, and consuming only 20% of our body energy (Clarke and Sokoloff, 1999). No wonder we have been trying to understand the brain and mimic its behavior!

The history of our attempts to understand the brain is as long as humans' history. Our first attempt to investigate how the brain works dates to ancient Egypt (Mohamed, 2008). But with Ramon y Cajal, we had a revolutionary contribution to understanding the nervous system's structure. For the first time, we had printed images of neurons and the knowledge that they were independent structures that are in touch without touching (Ehrlich, 2022). Since then, neuroscientists have studied the nervous system in all its aspects: structure, function, development, diseases, and more. We have been trying to map a human connectome and other animal brains, and advances in neuroimaging technologies and machine learning were (and still are) crucial for that (Morita et al., 2016; Nunes, 2021; Witvliet et al., 2021). These advances allow us to identify diverse brain structural connectivity patterns that give rise to a wide range of processes.

Despite the tremendous progress in computing, in AI, in neurotechnologies, and theoretical neuroscience, we still don't really know if the brain functions are defining its architecture or if the architecture of the brain gives rise to specific functions. We understand that computation emerges from the interaction of neural areas (McCulloch and Pitts, 1943; Gilson et al., 2015; Mill et al., 2017; Reid et al., 2019; Ito et al., 2020) and that the structural wiring of the brain (either locally or globally) is highly correlated with the brain function such as memory, vision, and motor control (Bullmore and Sporns, 2009, 2012; He and Evans, 2010; Kaiser, 2011; Meunier et al., 2010; van den Heuvel and Sporns, 2013; Lynn and Bassett, 2019).

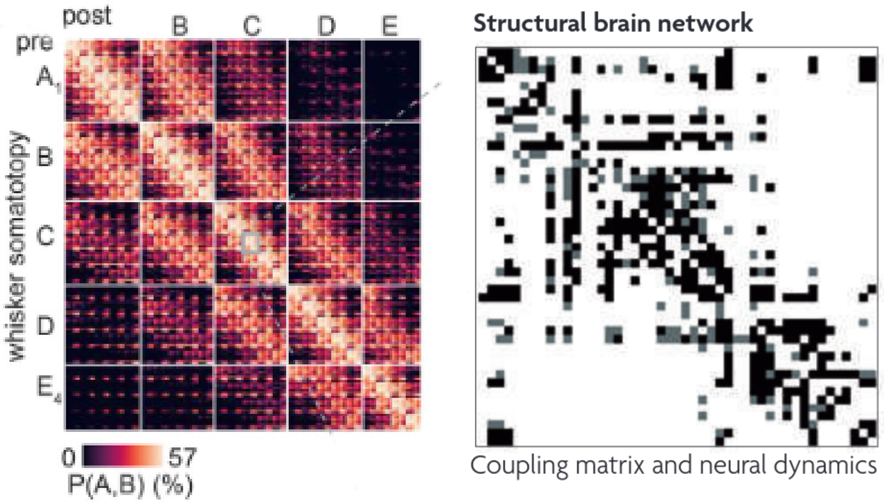
Still, standard computing technologies do not use the wiring and connectivity patterns that the brain does. We can recognize in our computers a similar set of high-level structures to what the brain has, such as cameras and computing vision engines that are equivalent to our eyes and visual cortex. However, the electronic implementations of such structures are not brain-like. This gap opened up a new field, neuromorphic engineering, where the development of circuits that share physical properties with biological nervous systems allowed fewer transistors than digital approaches to emulating neural systems (Mead and Ismail, 1989). One key characteristic of neuromorphic engineering is to understand how the morphology of individual neurons and the topology of neural networks affect the representation of information and computation. And from there, to design brain-like devices and systems that can provide robust and efficient computation using low-power and massively parallel analog Very Large Scale Integration (VLSI) circuits (Mead, 1990, 2020).

In this Thesis, following the neuromorphic computing approach, we explore the topology of neural networks to advance the design of new brain-like devices. We look at biological brains and propose brain-inspired architectures and strategies to change how to place and route networks on those devices. We focus on small-world networks, as a pattern of connectivity found not only in biological neural networks but also in Artificial Neural Networks (ANNs) that perform brain-like computation.

1.1. Connectivity in the brain

In the brain, we find diverse patterns of structural connectivity. Those different patterns support a wide range of cognition and behaviors. In a recent review, Lynn and Basset discuss how network structure relates to function and control, and, at first glance, the wiring of the brain shows itself far from homogeneous (Lynn and Bassett, 2019). Some aspects of brain networks are already well-studied and defined. For instance, large-scale functional brain networks express characteristics of modular, small-world, heavy-tailed, and metabolically constrained organization (Lynn and Bassett, 2019). In other words, brain networks, or their basic network composition, show a modular community structure, or cluster of neurons, that are internally densely connected and externally weakly coupled, with short path length and high clustering (Bullmore and Sporns, 2009; Sporns and Betzel, 2016).

Modular networks give numerous advantages for nervous systems, and the robustness of "evolving" or adapting each module independently without losing a global functionality certainly is a



(a) (b)

Figure 1.3.: Connectivity matrices. A connectivity matrix is a square matrix that shows the influence a neuron (or area) exerts on another. (a) Matrix of connection probabilities between excitatory neurons in the connectome of rat barrel cortex (every row and column) sorted by the somatotopic location (blocks from A to E) (Udvary et al., 2020). (b) Structural brain network of macacque cortex (Bullmore and Sporns, 2009). The surface of the macaque cortex was subdivided into 47 areas (shown in the rows and columns), and a structural brain network linking these nodes was compiled from anatomical tract-tracing data.

significant one (Meunier et al., 2010). In the brain, information encoding is achieved via population coding, using spatiotemporal patterns of activities instead of relying on the activity of single neurons (Averbeck et al., 2006; Sakurai, 1996; Pouget et al., 2000). This spatiotemporal activity of populations is highly dependent on their connectivity and physical layout (Horvát et al., 2016), which affects the brain's computations. Notably, the use of modular networks with high clustering of connections between nodes allows for locally segregated processing with low wiring cost (Meunier et al., 2010). Indeed, biological neural networks are often highly recurrent, with dense connections for nearby neurons and sparse connections to specific or far away neurons (Laughlin and Sejnowski, 2003). They often present an exponential decay in the number of connections with increasing distance.

Figure 1.3 shows some examples of connectivity matrices from biology. A connectivity matrix is one way to represent information about brain connectivity. If a connection between two areas (populations or nodes) is found, then a value 1 is added to the place where the areas meet: the row and column intersection. If no connection exists, then the value is set to 0. In Fig. 1.3 we can see dense connections over nearby areas and sparse connections with increasing distance, reinforcing the modular structure of brain networks.

1.2. Connectivity in the neo-cortex

This modular organization is found not only between brain areas but also within cortical areas. The microcircuit of the cerebral cortex in the mammalian brain is considered an essential element in generating our impressive capabilities: it is in the cortex that voluntary control of behavior and cognitive processes originate (Harris et al., 2019). We know cortical regions are organized into columns and layers, and most connections between layers show columnar functional organization. Also, columns interact through long-range connections laterally. This pattern of connectivity is found incredibly conserved across regions, suggesting a canonical microcircuit (Douglas et al., 1989; Douglas and Martin, 2004; Binzegger et al., 2005).

And although we don't know yet what the function of such a circuit is, we can describe some of its characteristics: neurons receiving direct inputs do not send axons outside the local region, and neurons that are driven by the input layers form long-range connections within their layer and also outside their region (Hawkins et al., 2017; Douglas and Martin, 2004).

Those characteristics lead to basic similarities among different areas, and finding these macroscopic characteristics is a step toward understanding how the networks generate functions. Also, biological implementations of neural networks offer problem-solving capabilities with critical energy and memory constraints. Computing technologies, and our AI systems, are also facing energy and memory constraints; we can now consider the solutions found by nature in our artificial systems. Brains avoid data transfer by having memory and computation colocalized. Biology offers us an inspiration to perform computation exploiting the physics of the system and with new paradigms.

1.3. Connectivity in electronic circuits

The classic von Neumann architecture uses separated structures for memory and processing. Any operation performed on it is based on a precise set of instructions executed to process data. To do so, it is required that both the data and the instructions are stored in the same memory area. The data movement between the physically separated processor and memory creates a bottleneck where latency is unavoidable.

Many approaches are trying to increase the parallelism capabilities of classical computers to decrease latency, for instance, by increasing the number of cores. However, the processing system is not the only component to determine the overall computational performance of a computer. The memory system used can represent, in fact, one of the significant performance bottlenecks in classical computers. Besides the latency, classical computers and machine learning techniques demand high power to perform computation. The primary source of power consumption is due to data movement from memory (especially the off-chip Dynamic Random Access Memory (DRAM) access) to the processing unit (Horowitz, 2014; Boybat Kara, 2020). Not surprisingly, many approaches have been proposed to reduce this data transfer, for instance, caching, multi-threading, and even different types of RAM computing and photonic devices (Goodman, 1983; Horowitz, 2014; Keckler et al., 2011; Ríos et al., 2019; Verma et al., 2019).

In recent years, one way to increase performance and reduce power consumption has been to use System-On-Chip (SOC). In SOC, we can also reduce the area of a normally multichip design onto a single processor that uses much less power than before. This is possible because we use specialized computing blocks for specific compute-intensive workloads (Hertz, 2022). They have been used in most portable technologies: mobile phones, tablets, cameras, and many other wireless devices. However, although moving towards reducing power and increasing performance, SOC does not solve the problem of data transfer. Either in multi-chips or SOC, data flow control between memory and processing blocks is a non-trivial issue.

In an attempt to mimic brain behavior, we shift our computational substrate from the classical von Neumann architecture to neuromorphic computing. This alternative approach is known as in-memory computing and creates an energy-efficient solution.

1.4. Connectivity in neuromorphic processors

In in-memory computing, the data is not stored anymore in a separate memory. It works by eliminating all slow data accesses and relying exclusively on data stored in RAM, avoiding input/output operations. The hardware is used as a memory to store information *and* to perform computation using the physics of memory devices at the same time (Boybat Kara, 2020). However, with still limited memory available, the connectivity on-chip must be carefully designed.

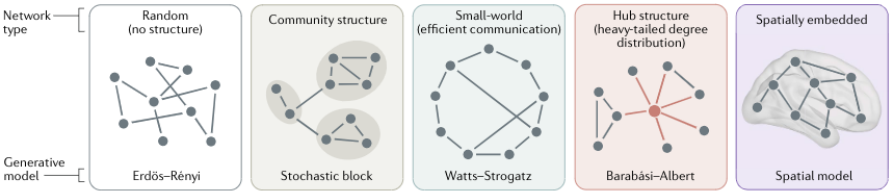


Figure 1.4.: Looking at the brain's wiring, we can see it is far from homogeneous. Structural brain networks can be characterized by mathematical models ranging from random to spatially embedded networks. Figure from (Lynn and Bassett, 2019).

The Network-on-Chip (NoC) architecture has been used to perform the communication of massively parallel systems, minimizing latency and router congestion. The communication process among nodes in a NoC system is based on a network of routers (Daneshtalab et al., 2010).

In this new architecture, the physical layout and connection between nodes (or cores) form the network's topology. The topology is a fundamental aspect of NoC design, and it is responsible for the overall cost and performance of the network. The topology influences the latency and power consumption because data will travel through the routers defined in the NoC. Therefore, the design of a reasonable NoC topology is crucial to system scalability.

Interestingly biological implementations of neural networks also have binding energy and memory constraints. The brain's typical connectivity pattern is understood as being encoded in part by genetic compatibility. And, the genome's information capacity limits the construction of biological neural networks. It is understood that the brain's precursor cell can have less than $1Gb$ of information, orders of magnitude small to encode the connectivity for billions of neurons explicitly. A naive encoding of this connection matrix (i.e., explicitly listing all the connections) would require at least $10TB$ for a mouse brain (Kerstjens, 2022).

Thus, it is safe to assume a direct mapping between neurons and their connectivity. This gives rise to a specific network encoding, meaning that the brain does not create random connectivities; instead, genetic interactions produce a stereotyped network (Barabási and Barabási, 2020; Barabási and Czégel, 2021; Kerstjens, 2022). More than that, physical laws are apparent in brain connectivity: the brain needs to minimize the wiring between neurons since it is constrained in a 3D shape (Horvát et al., 2016). Biological neural circuits need a massive amount of fast and durable connections. The communication between neurons through the axons (wires) needs to be optimized to minimize delays and the length of its wires while maximizing the density of synapses.

Here, we want to exploit the solution found by nature to guide the design of NoC for neuromorphic hardware to improve its scalability.

1.4.1. Physical substrate

The brain is physically constrained in a 3D space. The wiring of networks in the brain uses physical connections: the synapses support the propagation of information between neurons, and the white matter tracts define communication pathways between regions (Lynn and Bassett, 2019). The analysis of this physical structure exposes a wide range of topological organizations in brain networks, such as community structures, small-worldness, and hub structures, see Figure 1.4. Nature has to solve the same problems we are trying to solve while designing silicon chips to achieve brain-like computation.

Neuromorphic chips have essentially a 2D-substrate where we design the NoC to implement our neurons and connectivity. Currently, to fit large neural network models into these accelerators, we use multi-core architectures (Moradi et al., 2018; Merolla et al., 2014; Akopyan et al., 2015; Davies et al., 2018; Furber and Bogdan, 2020). In these architectures, each core either *emulates* with analog circuits (Moradi et al., 2018) or *simulates* with time-multiplexed digital circuits (Akopyan et al., 2015; Davies et al., 2018; Furber and Bogdan, 2020) neuro-synaptic arrays in which both the synaptic weight matrix and the network connectivity memory blocks occupy a significant proportion of the total layout area. Finding trade-offs to optimize both weight-matrix and connectivity/routing memory structures in multi-core neuromorphic processors can significantly impact their total chip die area and the size of the networks they can implement. Specifically, by focusing on small-world network connectivity, we can minimize memory consumption requirements while still enabling the design of neural network architectures that can solve a wide range of relevant "edge-computing" problems, i.e., the types of sensory-motor processing problems that animals must solve in the real world.

1.4.2. Creating connections

The encoding of a system as a network and the quantitative assessment of its topology can provide essential insights into its function. The analysis of a brain as a network can be done on different levels, for instance, structural or functional. Nodes can be seen either as neurons or cortical areas and edges as axons or fiber tracts. Modeling the brain as a network allows us to use new techniques and abstract away non-important or non-understandable information. The communication among specialized brain regions and the integrative functions performed locally creates a substrate that allows the creation of specific cognitive states. The combination of architectural inter-area features with the local plasticity within areas gives the brain the power to perform complex functions.

We can extend our biological observations to our NoC design. The nodes on a chip share information through a router network. As previously explained, this network's topology greatly impacts its scalability. By focusing on small-world network structures, we can create levels of communication between routers that are optimized for such structures, thus, following a specific topology.

The design of a new NoC architecture brings us the challenge of new algorithms for the placement of the network. A targeted design introduces more constraints than when we work with a general approach design.

1.4.3. Selected networks

Since biological neural networks follow a structured motif, we performed a systematic analysis to find the patterns and motifs in ANNs used in neuromorphic computing applications.

We surveyed and identified the most common network architectures in use (Donati et al., 2018, 2019; Risi et al., 2021; Krause et al., 2021; Kreiser et al., 2020) (see also Appendix A).

We found that the Winner-Take-All (WTA) architecture is among the most commonly used in the community. WTA networks, in their simplest abstract form, consist of an excitatory unit (either a neuron or a population) that projects to an inhibitory unit. This inhibitory unit, in turn, provides recurrent inhibitory feedback to all excitatory units (Feldman and Ballard, 1982), as shown in Figure 1.5 (Watts and Strogatz, 1998).

This modular structure with sparse connections between densely connected excitatory (E) and inhibitory (I) populations is also commonly used in other networks, such as reservoir and relational networks.

Furthermore, these networks are being used in a wide range of applications, e.g., filtering out the noise, supervised and unsupervised learning, increasing discrimination in the inference of a classifier, creating relations between populations, and representing variables with population coding (Donati et al., 2018, 2019; Risi et al., 2021; Krause et al., 2021; Kreiser et al., 2020).

We can optimize hardware parameters and algorithms by focusing on a specific connectivity scheme while not compromising the applications developed.

Mainly, by being a type of connectivity we find in biological brains, we gain insights into how to construct our applications and artificial networks. To evaluate if this modular, small-world structure found in biological networks impacts the performance of ANN, we constrained connections spatially during the learning of a task. We imposed a penalty term in the loss function for training, such that long axons (or connections between distant neurons) would be more costly.

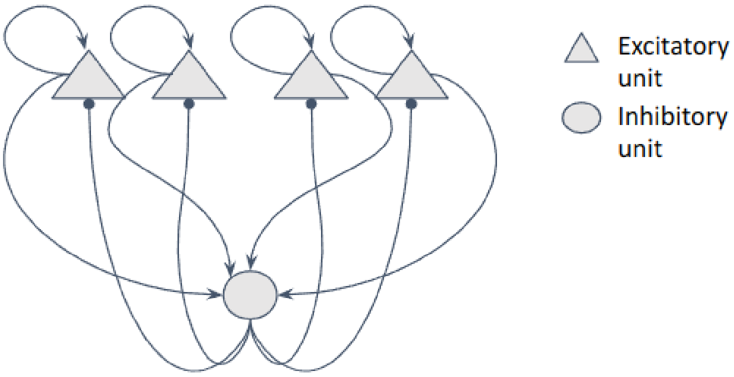


Figure 1.5.: Schematic of a basic WTA network. A set of excitatory units interact with an inhibitory unit.

Our findings indicate that we can push ANNs to prefer short-distance connections over long-distance ones and that accuracy is not highly impacted (see Appendix C), confirming that the small-worldness of networks is beneficial for Machine Learning (ML) applications (Zheng et al., 2010; Li et al., 2013).

1.5. Neuromorphic ecosystems

Standard computing technologies have matured in the last sixty years because of the shared layers of abstractions, also called the computing stack. This common ground creates an interface between the devices and the end-user applications. Every layer in this stack provides primitives that are used by the neighboring layers, allowing parallel development of each layer while maintaining operability and system compatibility.

Recently, neuromorphic systems have been growing and becoming more and more complex. The development of a neuromorphic computing stack is essential to demonstrate neuromorphic hardware as a viable computing platform.

Neuromorphic systems are implemented by *neuromorphic hardware*, *system software*, and *application software*. Nowadays, the integration between layers needs to be better defined, mainly because neuromorphic computing structures target a much more specialized range of algorithms. This specificity in applications, hardware, and algorithms makes it even more essential that application developers and end users work together from the start of any research development. Figure 1.6 shows how we envision the computing stack for neuromorphic computing using the layers of a neuromorphic system.

The research community studies many emerging devices and materials for neuromorphic hardware implementation. And, as a new and active research field, there are few application-ready, full-scalable neuromorphic systems available to the community. While they are an excellent platform for efficient real-time simulation of neuronal dynamics and synaptic transmission, the configuration and tuning of such systems are typically demanding. They must be facilitated to allow non-hardware experts to use them. To address these issues, efficient ways to design and use neuromorphic hardware are needed.

The most common approach to emulating neural networks with neuromorphic hardware is to expose its architectures and details. The specific knowledge requirement needed to use these systems led to a significant amount of software design and implementation to hide these architectures and facilitate the development of applications to solve real problems. Each neuromorphic chip has different specifications; however, a familiar ecosystem, including a high-level program interface (either by Graphical User Interface (GUI) or console for a high-level language), a com-

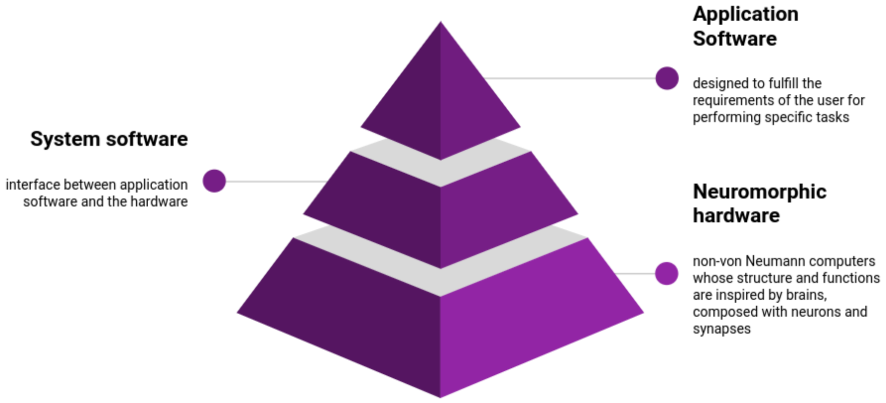

Application Software
designed to fulfill the requirements of the user for performing specific tasks
System software
interface between application software and the hardware
Neuromorphic hardware
non-von Neumann computers whose structure and functions are inspired by brains, composed with neurons and synapses

Figure 1.6.: Schematic of the neuromorphic computing stack. Neuromorphic systems include neuromorphic hardware, system software, and application software. The integration between the layers and a well-defined set of primitives will provide the means of developing and evolving neuromorphic systems at a faster pace. A set of supporting software is necessary to make the most of neuromorphic hardware. Here we divide them into system software and application software. System software is the interface between the hardware and the user applications. And application software is built to run specific tasks for the user.

piler, and a visualizer, helps to accelerate applications' development.

More than emulating the brain, we need to find a different abstraction to get the necessary insights and translate them into machines. Nevertheless, even without entirely defining this new abstraction, it is clear that we need new ways to program and develop this hardware. Consequently, supporting software has been developed to make neuromorphic hardware more practical.

1.5.1. Neuromorphic hardware

Following biological plausibility, SNNs were developed to perform parallel computation. These networks are different from generic ANNs in that they model the action potential generation mechanism of real neurons to produce output spikes in response to integrated input spikes, preserving the information on the timing of the input signals. The output of a neuron in an SNN is dependent on previous stimuli, and the neuron only fires after crossing a threshold. Neuromorphic hardware has been designed to exploit SNNs' parallel computation abilities.

Neuromorphic processors are a class of AI hardware accelerators that implement computational models of SNNs adopting in-memory computing strategies and brain-inspired principles of computation (Roy et al., 2019; Chicca and Indiveri, 2020; Sebastian et al., 2020). In our brains, memory and computation are colocalized instead of having processing units separated from memory, as in classical von Neumann architectures, as seen in Figure 1.7.

In an attempt to mimic the brain, neuromorphic electronic circuits were created, offering information processing on-demand (driven by events) in an energy-efficient and asynchronous way, analogous to biological systems (Liu et al., 2014). They emulate the structure and function of our neural system, not only to exploit the parallel computation but also to understand their computational properties. Neuromorphic hardware represents a very promising approach, especially for edge-computing applications, as it has the potential to reduce power consumption to ultra-low (e.g., sub milliwatt) figures (Covi et al., 2021), thus easing AI computing workloads. However, the requirement of SNN hardware accelerators to store the state of each neuron, combined with their in-memory computing circuit design techniques, leads to very large area consumption figures, which limits the sizes and numbers of parameters of the networks that they can implement. New materials and devices can overcome such limitations. However, for now, these hardware

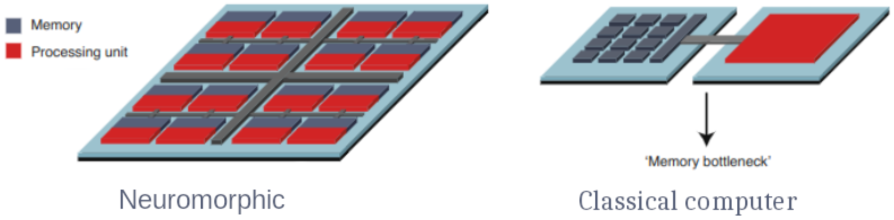


Figure 1.7.: Schematic of the neuromorphic (left) and the von Neumann architecture (right). In neuromorphic hardware, memory and processing are co-located, avoiding the memory bottleneck. Figure adapted from (Zhang et al., 2020).

constraints also imply restrictions to the models and, consequently, to the applications that can be emulated on hardware.

While neuromorphic hardware supports SNNs, the differences between the hardware and the SNN representation often complicate using the hardware capabilities. The challenge now is how we can use and program this new hardware and how this can help us get insights into how brains perform computation. A new hardware platform brings an opportunity to create new ways to think about computation (Backus, 1978).

To tackle this, besides creating new hardware, the community has been developing new programming languages (Eppler et al., 2008; Davison et al., 2008; NES, 2016; Amir et al., 2013) and new algorithms (Bellec et al., 2020; Nicola and Clopath, 2017; Lillicrap et al., 2016).

1.5.2. Application software

Application software is a program created to perform a specific task. It is user-specific and is designed to meet the requirements of the user. The application software is often uncoupled from the hardware, and the hardware resources can be ignored by adding suitable software abstraction layers.

In this Thesis, we define application software for neuromorphic hardware as any software that helps the SNN developer to design, parameterize and prepare the SNN to be deployed in the neuromorphic architecture. This way, the application software is the primary tool of the SNN developer. Although SNNs can, in theory, reach the same accuracy as classical ANNs (Maass and Markram, 2004), we haven't been able to exploit yet their computational abilities in machine learning tasks. One big reason for it is related to how new the field is; thus, we do not have standard tools and algorithms to train and leverage the advantages of SNN.

In neuromorphic systems, these tools do not run over neuromorphic hardware. They can be uncoupled entirely with any neuromorphic hardware instance, use a hardware simulator or have hardware as a backend.

It is common to see SNN developers using their own scripts and code. And not surprisingly, there are a few application software systems that offer a common way to handle the SNN design, e.g., Brian 2 (Stimberg et al., 2019), Teili (Milde et al., 2018), Nest (Gewaltig and Diesmann, 2007), and Nengo (Bekolay et al., 2014).

They all have the same goal: to provide a more accessible infrastructure to design SNN.

1.5.3. System software

System software is a set of programs that control and manage the operations of the hardware. They are designed to manage the system's resources, like memory, processes, and security.

They are the interface between the application software and the hardware itself. It represents the abstraction layer of the hardware. For neuromorphic systems, it can be a compiler or another type of software that controls how the SNN will run based on the configuration of hardware parameters.

One common approach to developing system software is to divorce it from the hardware and create software that can potentially be used by multiple (different) hardware system or in a way that will not impact the hardware development. This approach is known as *platform-based design*. Another approach is the *hardware-software co-design* approach. Here, hardware and software are designed simultaneously to exploit the best of both, even though another architecture can not easily use the same software. In this work, we use this second approach.
Together with the development of this work, we designed a new system software to facilitate the use of the Dynamic Neuromorphic Asynchronous Processor (DYNAP) chip. CortexControl (CTXCTL) (Cor, 2020) offers a Graphical User Interface (GUI) and a Python Console to control and execute experiments on neuromorphic hardware. More detailed information about the software can be found in Appendix B.

1.6. Thesis overview

This introduction outlined how brains and computers are related and how we can emulate the brain in electronic circuits and neuromorphic processors. We highlight a connectivity pattern found in biological and artificial neural networks. We also introduced the concept of neuromorphic ecosystems and the computing stack, where integrating neuromorphic hardware, application software, and system software is necessary to advance the neuromorphic field at a faster pace. In Chapter 2, we explain in more detail the system software and compilers, including platform-based design, hardware-software co-design, and the process of routing and placement of a network. In this Thesis, we present a new approach to reducing the use of memory on hardware. The reduction in memory can lead to a reduction in chip area, allowing the scalability of neuromorphic devices. To design our routing scheme, we focus on the network's topology and exploit the brain's physical constraints, reinforcing the idea that (physical) distance matters. In Chapter 3 we describe the router communication in our NoC for small-world networks. This specificity allows us to significantly reduce the amount of memory needed in the hardware and, thus, reduce the total chip die area. However, the fixed topology brings more constraints to the placement of networks. In Chapter 4 we present an algorithm that can deal automatically with the constraints introduced to facilitate the use of this new hardware structure. Our placement algorithm consists of three parts: neuron placement, distance calculation, and synapse assignment. In Chapter 5 we discuss the results of our experiments and show memory comparison with other major systems. Additionally, we present a real-case application scenario, where we map an Recurrent Neural Networks (RNN) into our hardware design. The Thesis concludes with Chapters 6 and 7, where we reflect on the research developed, its overall impact, implications, and some limitations and recommendations for future research.

"In the long history of humankind those who learned to collaborate and improvise most effectively have prevailed."

Charles Darwin

2. System software and compilers

As explained in Chapter 1, the system software is the set of applications that control and manage hardware operations. For neuromorphic systems, we define them as compilers or another control for how the SNN will run based on the hardware configurations.

Neuromorphic hardware configurations are based on modeling biological sensors and neural networks. Here we focus on multi-core architecture, where each core contains neurons and synapse circuitries. The most common implementation of a core is to build the circuitries as crossbar arrays. A crossbar array is a matrix-like structure where neurons can be easily fully connected to each other (see Figure 2.1).

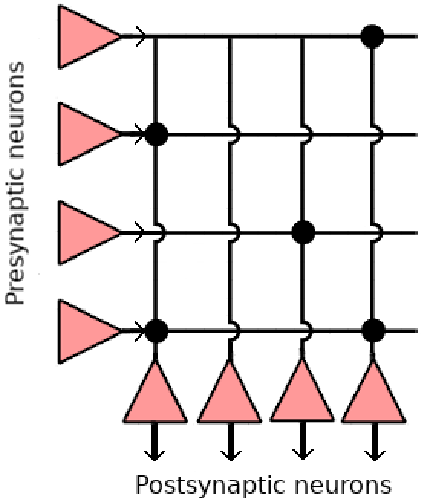


Figure 2.1.: Schematics of a crossbar array. A crossbar is a matrix-like structure where neurons are hardwired. Black dots mark points where pre- and post-synaptic neurons have an active connection. Every junction requires a single bit to create a connection between neurons.

In a crossbar, the memory elements are built at the crossing points of horizontal and vertical access lines. They provide an easy way to create high-density memory and storage. A N-way fan-out can be supported within a core simply by the actual wires, and it is as memory and energy efficient as it can be. However, this has its own drawbacks: crossbars are not scalable to a large number of neurons. Not only would a large number of neurons require a larger area, but also, the sneak paths (i.e., the undesired paths for current, parallel to the intended direction) formed by unselected points in parallel with a selected point can lead to issues during reading and writing operations.

The advent of nano-scale memristive devices and even 3D-VLSI technologies can mitigate some of the problems by enabling the construction of dense crossbar array structures for storing the weight matrices (Sebastian et al., 2020). Still, scaling up networks and connectivity is a problem of fundamental nature.

To provide the same order of fan-in and fan-out seen in biology, a communication protocol called Address-Event Representation (AER) was created. AER is used to time-multiplex neuron outputs onto a shared bus, as depicted in Figure 2.2.

This way, the number of wires needed to connect neurons (and thus, the area) can be reduced

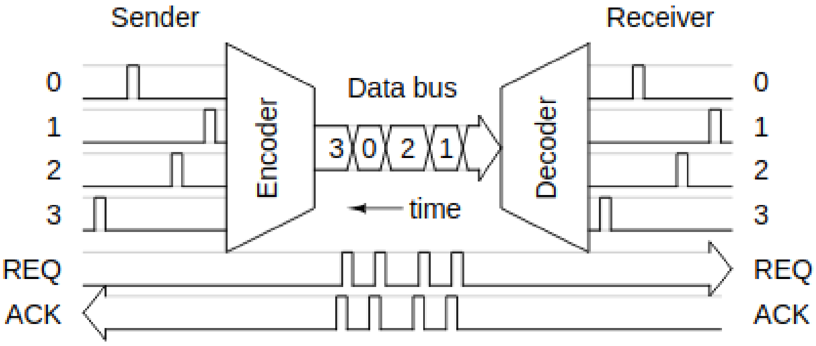


Figure 2.2.: AER. The communication between pre- and post-synaptic neurons happens through Address-Event (AE) sent over a shared bus. With the use of AER, the number of wires needed to connect neurons is reduced from N (as in the crossbar array) to $\sim \log_2 N$, where N is the number of neurons. Figure adapted from (Vogelstein et al., 2003).

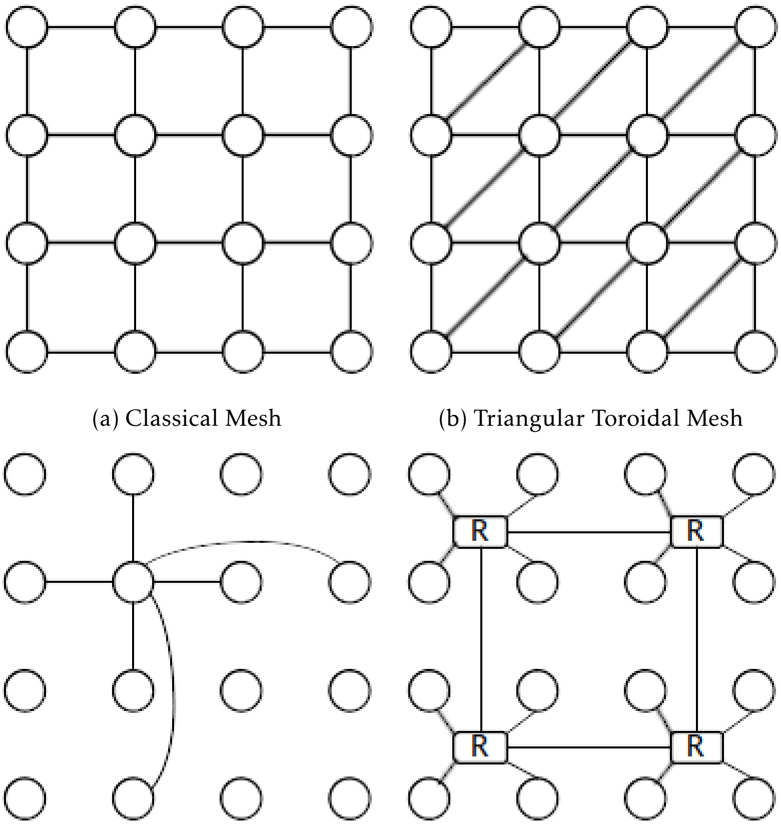
from N, with one active at a time, to $\sim \log_2 N$ wires that are simultaneously active. The key idea of the time-multiplex approach is to exploit the fact that electronic communication is several orders of magnitude faster than the firing rates of biological neurons, trading-off space for speed of processing to address the scalability and connectivity of a large number of neurons (Mahowald, 1994).

While trying to mimic the brain, in hardware implementations, it is not feasible to have neurons connected directly physically. Multi-core neuromorphic processors usually use NoC designs for managing the communication of neurons between cores. And different spike routing architectures have been proposed according to their applications. The most common topology and routing structures of NoC used in current state-of-the-art neuromorphic hardware are shown in Figure 2.3.

Classical mesh architectures (Davies et al., 2018; Merolla et al., 2014) represent an easy way to build large-scale systems. However, when the size of NoC increases, the required hardware area increases considerably, reducing system scalability. In flattened butterfly architectures (Chen et al., 2019), neuron cores belonging to the same row and column can communicate directly with lower routing latency. However, this architecture also brings the disadvantage of significant area cost and poor multi-casting support. In (Park et al., 2016), the authors proposed a hierarchical architecture that overcomes some of these disadvantages by using off-chip DRAM to store the routing lookup table, which significantly increases power consumption. Current methods for saving power adopt in- or near-memory computing strategies. However, when on-chip memory is used to store configurable neuron connections, the required hardware area increases proportionally with the number of neurons and synapses.

Every neuromorphic hardware needs system software to configure and program the network connectivity onto the chip. System software can separate the hardware requirements and constraints from the SNN models, offering a translation machine from the model into the hardware. This translation needs to take care of variability in the SNN models (architectures, neuron types, synaptic information, and more), availability of resources on the hardware, and its configuration (connections among neurons).

Therefore, the abstraction level used for programming the models is the bridge between hardware and software. System software is an essential part of the computational stack. And the development of such a layered structure allows users not to be concerned about how the hardware is structured and how resources are allocated. Of course, knowledge about hardware specifications can be beneficial when designing SNNs. However, having hardware requirements as a constraint for model development can slow down our process of getting insights about the brain. We can develop a more efficient way to address the application demands when we provide a well-defined



(a) Classical Mesh

(b) Triangular Toroidal Mesh

(c) Flattened butterfly Mesh

(d) Hierarchical + Mesh

Figure 2.3.: Different types of NoC architecture. Every circle represents a core in the hardware. Links between cores represent the direct routing path allowed in each architecture. (a) Classical mesh design. Mesh design allow an easy way to build large-scale neuromorphic hardware, but it has high routing power comsumption, area cost and latency (Davies et al., 2018; Merolla et al., 2014). (b) In triangular toroidal mesh there are additional routing channels which increases the bandwith of the network (Furber et al., 2014). (c) In flattened butterfly mesh architecture the cores in the same row and colum can be reached directly (Chen et al., 2019). (d) A mixed hierarchical and mesh architecture can be used to improve power consumption, area costs and latency (Moradi et al., 2018).

computational stack, from applications to the underlying hardware.
The development of such system software follows, most commonly, two approaches: *platform-based* and *hardware-software co-* design.

2.1. Platform-based design

Platform-based designs have software development divorced from hardware (Sangiovanni-Vincentelli and Martin, 2001; Vincentelli, 2002). It is essential to clarify that this does not mean that the software is unaware of the hardware, but their development are orthogonal.
In software engineering, a system is considered orthogonal if changing one of its components only changes that component's state. Having the software and hardware development orthogonal to each other means that software does not impact hardware development, and vice-versa.
This independency allows the exploration of alternative solutions, for instance, by considering unlimited hardware resources. Moreover, it allows the provision of part of the system software by optimizing one or more of the diverse hardware parameters: number of fan-in (incoming) and fan-out (outgoing) connections, memory capacity, connectivity, bandwidth, power consumption, etc.
Developing software not attached to hardware constraints also allows the reuse of the same system software for many different hardware platforms.
Several system software ignores some hardware details, allowing the mapping of somewhat generic networks to the hardware being used, or even simulating hardware being used (Sahu and Chattopadhyay, 2013; Das et al., 2018; Bouvier et al., 2019). Other focus on specific hardware characteristics to provide a more optimized mapping (Urgese et al., 2016; Balaji et al., 2020b). Some work does not provide a complete system software but focuses on parts of it: either the partition of networks to suit some hardware structure or the placement of an already fitting network (Mysore et al., 2022).

2.2. Hardware-software co-design

In a hardware-software co-design approach, hardware and software are designed together to exploit their integration (De Michell and Gupta, 1997; Darwish and Bayoumi, 2005). Advances in technology are driven by innovation and by our expectations of functionality. More innovative electronic devices nowadays mean better hardware technology and better software. With hardware technology almost achieving a plateau, a co-design approach is a more promising way to guarantee innovation and increase chip functionality (De Michell and Gupta, 1997).
In the co-design approach, the software is tailor-made for the hardware platform. We can then evaluate the impact of hardware design decisions on real-world applications and algorithm performance. We can also provide feedback to neuromorphic devices, and materials researchers on the performance requirements, such as size, weight, and power, that fully digital, CMOS-based designs cannot meet.
A significant advantage of this approach is to create better hardware-software interfaces. However, to do so, it demands that software engineers are familiar with both hardware and software systems or that software and hardware engineers work closely together. Hardware-software co-design approach is a cyclic design methodology: the design of a hardware circuit and its function could be defined by the execution and analysis of a software program; and at the same time, the software could be modified given the hardware design. In neuromorphic applications, system software allows us to reprogram the connectivity on the chip to perform a different set of functions without changing the underlying hardware.
Figure 2.4 depicts the difference between a platform-based design and hardware-software co-design approaches.
The advances in electronic circuits and the expansion of software systems are reinforced by the availability of resources and the capacity to reuse parts of the systems as building blocks. Here, we will focus on two main aspects of what a system software should provide for neuromorphic

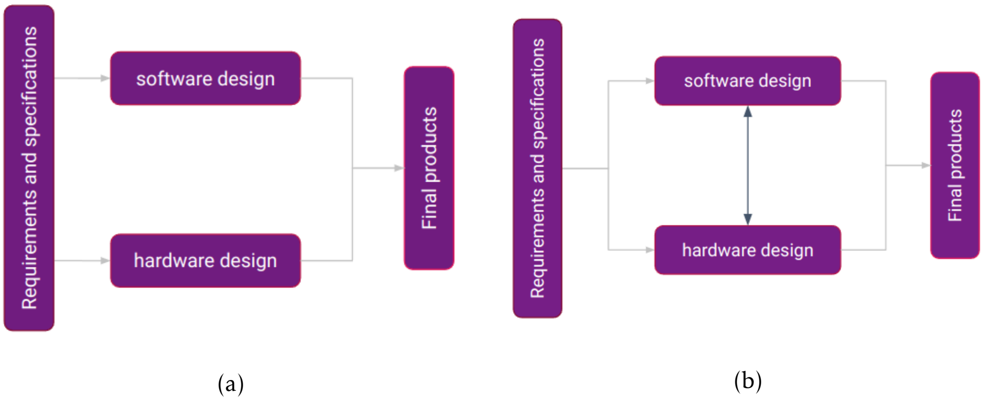


Figure 2.4.: Hardware and software designs. (a) Classical design flow of hardware and software development, in which they are designed independently, with only shared abstraction for compatibility. (b) Software-hardware co-design flow. In a concurrent development methodology, software and hardware are interlinked and dependent on each other.

hardware: placement and routing. The problem of placing neurons in cores and routing spikes among them must consider the hardware's core organization and routing scheme. The placement (also called mapping) is the process of taking the network's nodes and finding a physical distribution on the cores of neuromorphic hardware. Routing is to take a network topology (the connections between nodes) and define a path (or route) of communication on the hardware. Note that routing will depend on the placement strategy. Core organization imposes a constraint on the number of neurons that can be placed together. Routing tables impose a constraint in terms of the neuron fan-out and connectivity. Allocating routing and connectivity resources to allow arbitrary networks (e.g., with all-to-all possible connections) at scale is fundamental.

2.3. Routing schemes

Neuromorphic hardware mainly uses two routing schemes: *source-address* and *destination-address* routing (Zamarreño-Ramos et al., 2012). Figure 2.5 shows a simplified schematics of both routing types. The choice of the routing scheme affects the efficient implementation of the SNN.

2.3.1. Source-address routing scheme

In source-address routing, the spike is routed based on the source neuron information, i.e., the neuron that originates the event. When a neuron spikes, its address is sent together in the spike event to a router, and the router will send this information to all the synapses in the network. When a spike is received, the router searches for the source address in local memory. This memory codes all the operations that should be performed when a source address is received: forwarding the event to one or more output ports, routing it to the local processor, or both simultaneously. Every synapse has information on the address of its source connection. So, when a spike arrives, the synapse can make a matching comparison between its information and the address in the event, and decide whether it will react to the spike or not. Source-address routing can be advantageous for high fan-out networks (a large number of outgoing connections) when multicast is used. With multicasting, each router can copy a spike based on the information saved in its routing table. A look-up table can implement a source-address routing scheme either with an off-chip memory to store all the routing information for each source neuron or by using Content Addressable Memory (CAM) in each synapse. Accessing an external memory slows down the routing of

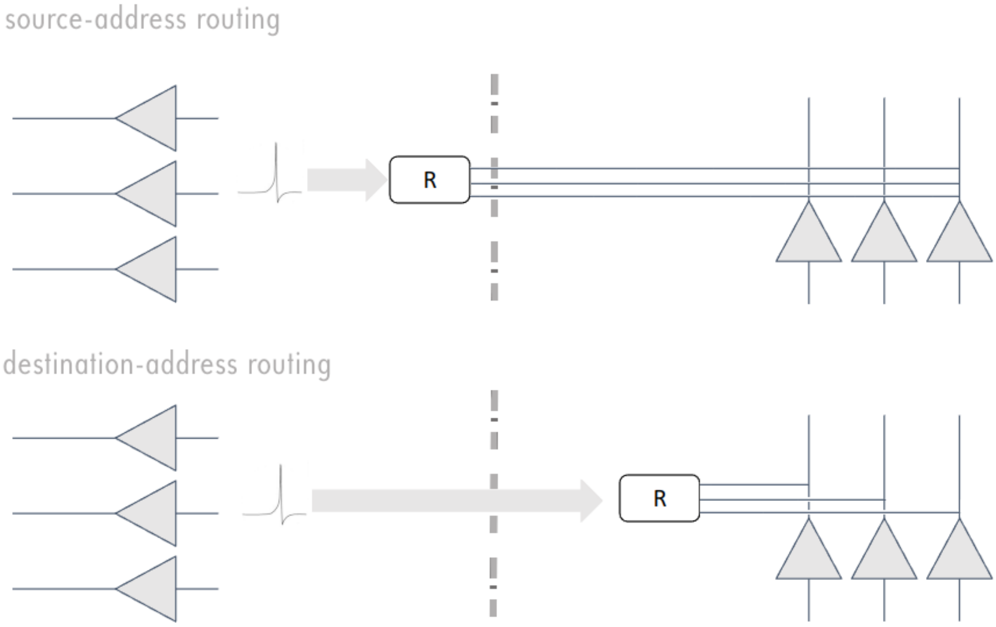


Figure 2.5.: The two basic routing schemes type for neuromorphic hardware. On top, the source-address routing scheme. When a source neuron generates a spike, its address is written in the AER package, and it is forwarded to all the destination neurons. Every destination neuron has a memory block with the address of its source neurons. If a match happens, the neuron will react to the spike. On bottom, the destination-address routing scheme. When a source neuron generates a spike, the destination address is sent in the AER package, and the router that now sits closer to the destination neurons can identify the specific destination where the spike needs to be sent. Source-address and destination-address define when the fan-out happens: if close to the source or the destination neuron.

spikes, giving an advantage for CAM. However, in any case, the number of connections that can be created is limited by the memory available to save the connectivity information.

2.3.2. Destination-address routing scheme

In destination-address routing, the spike is routed based on the destination neurons, i.e., the synapses that will receive the event. When a neuron spikes, it creates an event for every target synapse, and the router can identify the correct locations to forward it. Compared to the source-address routing, this avoids neurons receiving unnecessary spikes. Whenever a neuron receives a spike, it should react to it. In this case, every source neuron knows its target synapse and encodes its address as part of the spike being sent. This routing avoids the use of large off-chip memory. Each router is then responsible simply for redirecting the spike to specific locations given the information written in the event. However, the information about the connectivity needs to be stored at the source neuron. Destination-address routing can be advantageous for networks with low fan-out.

2.3.3. Mixed-address routing scheme

Another approach to route spikes in AER is to use a combination of point-to-point source routing and multicast destination-address routing. In this case, each source neuron stores the addresses of target cores (instead of specific target synapses). Then, the router receives the information about the core the spike needs to be sent and broadcasts it to all the neurons in that group. Every target synapse, in turn, has the information about its source neuron and performs a matched comparison to decide if it should react to a spike. This approach reduces the use of memory in the source neuron and the number of unnecessary spikes a synapse can receive.

Independently of the type of routing, it is necessary to save the addresses of the neurons in memory to know where to route the spikes (i.e., to form the network connectivity). In neuromorphic hardware, this can be done by using two main types of memory: CAM or Ternary Content-Addressable Memory (TCAM).

CAM A CAM is a memory that implements the lookup-table function in a single clock cycle using dedicated comparison circuitry (Schultz, 1997; Pagiamtzis and Sheikholeslami, 2006). It can store and query binary inputs, i.e., 0 and 1. In CAM memory, instead of retrieving data by accessing a memory address, the data is recovered by an exact match-based search of the data itself. The memory recovers the addresses where the data can be found. CAM is much faster than RAM in search tasks, however power consumption in CAM is still high in comparison with RAM of similar size (Jamil, 1997; Rajendran et al., 2011).

TCAM TCAM is a special type of CAM that allows a third state of "don’t care" in one or more bits of stored data (Rajendran et al., 2011; Perniola et al., 2018). Similarly to CAM, it also performs a search in a single clock cycle. The "don’t care" state increases the flexibility of searching where the data doesn’t need to be an exact match (Agrawal and Sherwood, 2006).

2.4. Placement approaches

Computational neuroscientists and SNN software developers describe networks in terms of neuron populations and connections without considering any hardware restrictions: the model is not coupled with the hardware. Nonetheless, hardware constraints play a significant role and impose severe limitations on the model that can be executed.

For instance, the most straightforward issue is when the model has more neurons or connections than the hardware can accommodate. In a *hardware-aware SNN modeling*, the number of neurons in the hardware would limit the SNN creation. Additionally, trying different hardware platforms can aggravate the burden on the SNN developer. Nonetheless, manual placement, known as hardware-aware modeling, is still a common and valid approach for small networks. However, each new platform has its own constraints, and updating the model to fit each platform is a time-consuming and error-prone task. An automatic way of doing it is essential to facilitate the use of neuromorphic hardware.

2.4.1. Hardware-aware SNN modeling

To develop an SNN to be "hardware-aware" means to take into consideration the hardware parameters or resources before SNN design. The hardware-aware modeling often can take two approaches: *a priori hardware knowledge* or *optimization of an existing network*.

The *a priori modeling* is the most accessible approach (from a hardware perspective): an SNN is designed from scratch with knowledge about hardware resources and parameters. And while it makes it easier for the SNN developer to try their network on the hardware, it limits the design choices.

In the second approach, the SNN is transformed or optimized to consider the hardware restrictions. Often, SNN developers define their own scripts to help with this task. But also, we can find work developed to make this process easier, faster, or less error-prone (Gopalakrishnan et al., 2019; Chowdhury and Shah, 2022; Milde et al., 2018).

Although there is no limitation on the design of the SNN, the conversion, either automatically or manually, to fit the hardware resources can lead to different behaviors that demand further tunning (Fang et al., 2019).

In this Thesis, however, we focus on the approaches that use software as part of the neuromorphic computational stack that is created to facilitate the placement and routing of SNNs. We are interested in the automatic mapping and routing for neuromorphic hardware.

2.4.2. Neuromorphic compilers

Developing a *neuromorphic compiler* is an emerging approach in the neuromorphic hardware community. A compiler is a program that can translate a high-level description (here, an SNN model) into a machine language (here, the chip-specific hardware configuration) without exposing the hardware structure.

Mapping an arbitrary SNN model onto neuromorphic hardware is not a trivial task. Partition and distribution of resources are challenging problems. In a more specific setting, partitioning a network into subgroups while minimizing some attributes can be seen as the K-way partition problem, which is an NP-Hard combinatorial optimization problem (Carlson and Nemhauser, 1966); thus, it is common to use approaches based on approximations through greedy algorithms, i.e., making an optimal choice at each step while attempting to find the optimal solution or other heuristics.

Every neuromorphic hardware needs its own software framework to place networks. The approach among them all is similar: partition of an SNN into clusters with several mapping strategies. Every strategy focuses on minimizing a specific attribute, such as energy consumption, latency, throughput, and many more (Das et al., 2018; Balaji et al., 2020b; Titirsha et al., 2021; Balaji et al., 2020a; Titirsha and Das, 2020; Lin et al., 2018; Balaji and Das, 2019; Song et al., 2020).

Besides placing an SNN to specific hardware, some approaches try to mitigate general constraints of neuromorphic hardware, such as limited precision or constrained computation, by decoupling the network from the target hardware (Ji et al., 2016, 2018; Rueckauer and Delbruck, 2016), for instance. And placing a network onto hardware also needs to consider how spikes are routed on the chip. TrueNorth (Merolla et al., 2014) and Loihi (Davies et al., 2018) use the destination-address routing scheme. SpiNNaker (Furber et al., 2014) and BrainScaleS (BrainScales, 2015) use the source-address routing scheme with the routing tables stored in a large off-chip memory. DYNAP (Moradi et al., 2018) uses on-chip hierarchical routing with a combination of point-to-point source-address routing and multi-cast destination-address to reduce memory usage.

	Loihi	TrueNorth	SpiNNaker	DYNAP
# neurons per core	1024	256	1000	256
# cores per chip	128	4096	18	4
core area	0.41 mm^2	0.0936mm^2	0.4 mm^2	9.6mm^2
chip die area	60 mm^2	430mm^2	102 mm^2	43.79mm^2
routing	destination-address	destination-address	source-address multicast	mixed
compiler	LCompiler	Corelet	PACMAN	-
technology	Digital ASIC at 14-nm CMOS	Digital ASIC at 28-nm CMOS	Digital, programmable ARM Cores 130-nm	Mixed-signal 180-nm CMOS
total neurons and synapses	130,000 neurons, 130 million synapses	1 million neurons, 256 million synapses	18,000 neurons and 1,800,000 synapses	1,024 neurons, 64,000 synapses

Table 2.1.: Comparison between some of the major neuromorphic chips.

Nevertheless, for any NoC architecture used, placing and routing a SNN onto neuromorphic hardware has begun to receive substantial attention (Galluppi et al., 2012; Amir et al., 2013; Lin et al., 2018; Ji et al., 2018; Fang et al., 2019; Das et al., 2018; Ji et al., 2016).

Every neuromorphic compiler integrates a different mapping technique in a hardware-specific framework. Also, independent compilers have been developed to focus on different structures or mapping strategies (Neckar et al., 2019; Ji et al., 2016; Benjamin et al., 2014). For example, Ji et al. (2018) introduces a compiler that transforms the network model into a computational graph until only a specific set of operations (defined by the hardware) is reached. Mysore et al. (2022) focus on the optimal way to partition a SNN to fit an extended version of hierarchical AE routing (HiAER).

The biggest producers of neuromorphic hardware provide their own compilers: it is essential to provide a good programming model and an easy-to-use tool to make the hardware accessible. SpiNNaker (Furber et al., 2014) uses PACMAN (Galluppi et al., 2012), BrainScales (BrainScales, 2015) uses PyNN (Davison et al., 2008), Intel's Loihi (Davies et al., 2018) uses LCompiler (Lin et al., 2018) and IBM's TrueNorth chip (Merolla et al., 2014) uses Corelet (Amir et al., 2013).

Here, we present the routing mechanisms and the placement approaches used by Loihi from Intel, TrueNorth from IBM, SpiNNaker from University of Manchester, and DYNAP from Institute of Neuroinformatics (INI). Table 2.1 shows a summary of their characteristics.

Loihi

Intel's Loihi (Davies et al., 2018) is a manycore digital neuromorphic processor with a programmable, on-chip learning engine for SNN. A Loihi chip contains 128 neuromorphic cores implementing 130.000 neurons and 130 million synapses. Each Loihi core contains 1024 neurons and uses a destination-address routing scheme with two megabits of Static Random Access Memory (SRAM) per core to keep the connectivity of the neurons in the core. The design of Loihi allows putting up to 4096 chips together, and spikes are communicated between the cores using events in an asynchronous NoC. Loihi doesn't depend on off-chip memory to create the connectivity but still uses a separate memory area per core to do so. This leads to various mapping constraints, including the maximum fan-in and fan-out connections.

Intel also developed a compiler for Loihi, LCompiler (Lin et al., 2018), allowing users to place SNN without knowing the hardware configurations. LCompiler receives the SNN specification from a Python code and produces a binary byte stream that maps the network onto the hardware in three steps: preprocessing, mapping, and code generation. In the preprocessing phase, the

SNN parameters are validated, shared configurations are extracted, and a connectivity matrix is generated. Then, a greedy algorithm maps the SNN onto neuron cores.

TrueNorth

TrueNorth (Merolla et al., 2014) is a digital multi-core neuromorphic chip developed by IBM. A TrueNorth chip contains 4096 neuromorphic cores, each containing 256 spiking neurons and 12.75 kilobytes of SRAM memory to store the connectivity. Each TrueNorth core communicates with its neighbors using a 2D-Mesh NoC. TrueNorth uses a destination-address routing scheme, and each event contains information on the destination core and axon. Within a core, a crossbar is used to implement synapses and spike communication. In TrueNorth, each core can support a fan-in and fan-out of at most 256 neurons. The placement of SNNs into the hardware is done using a programming paradigm called Corelet (Amir et al., 2013). Corelet builds and deploys SNN to their architecture. Since it is bounded to the hardware, it is used to build SNN using functional units, called Corelets, that are preconfigured. This way, the process of SNN placement is relatively direct, mapping their functional units onto the chip's cores.

SpiNNaker

SpiNNaker (Spiking Neural Network Architecture) (Furber et al., 2014) is a massively-parallel multi-core computing platform composed of general purpose ARM cores. Each SpiNNaker chip has 18 ARM968 cores processors embedded in a programmable NoC. Each processor contains a local 32 kilobytes instruction memory and 64 kilobytes data memory. Additionally, each SpiNNaker chip contains 128MB RAM shared by the 18 ARM cores. Each chip can simulate a few thousand simple neuron models, with approximately 1000 input synapses per neuron. The SpiNNaker architecture is highly scalable, from a single chip up to 65,536 chips. SpiNNaker uses PACMAN (Galluppi et al., 2012) to model SNN onto their hardware. PACMAN uses PyNN (Davison et al., 2009) as the higher-level interface of the system, where the model is created. Their software operates in three main steps: splitting, grouping, and mapping. First, it splits SNNs too large to fit into a single core into subgraphs. Then it groups these subgraphs to add as many subgraphs as possible into a single core. After that, it allocates neural groups to processors using greedy algorithms and calculates routing.

DYNAP-SE

DYNAP-SE (Moradi et al., 2018) is a mixed-signal multi-core neuromorphic processor with four cores, where each core implements 256 analog neurons arranged in a 16×16 crossbar. Each neuron has 64 programmable synapses leading to a max fan-in of 64 connections and a maximum fan-out of 4k connections. DYNAP-SE implements a two-stage routing scheme. It uses a combination of point-to-point source routing and multicast destination-address routing. Each neuron has a *source* memory to store the addresses of the destination cores (maximum of four in DYNAP) and a *target* memory to keep the tags that its (64) synapses use to accept the spike from the matching sender neuron address. There are fewer cores than neurons, so the source memory is smaller than the destination one. Figure 2.6 shows how DYNAP-SE handles the routing.

DYNAP-SE does not have a compiler. The mapping of a SNN to a chip is done manually by the SNN developer.

2.5. Summary

Those approaches make it clear that compilation and automatic distribution of resources are the next challenges on neuromorphic hardware, and it is a required task. Although every neuromorphic hardware and simulator provides tools with specific mapping techniques, and the work

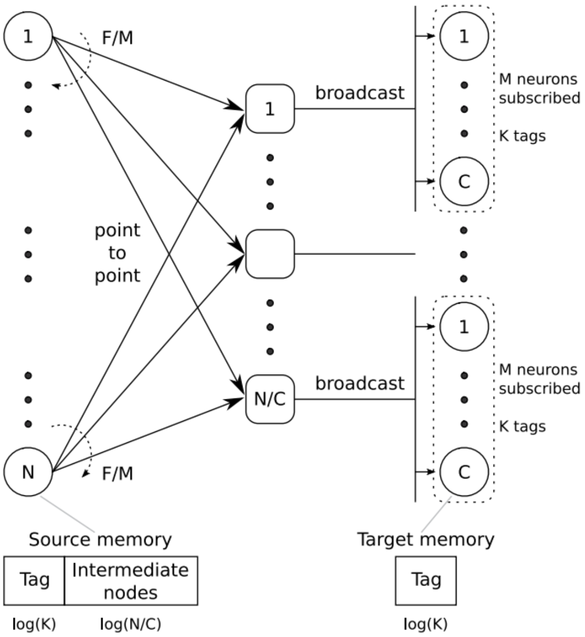


Figure 2.6.: Two-stage tag-based routing scheme used in DYNAP-SE. Figure adapted from (Moradi et al., 2018). Every time a spike is generated by one of the *N* neurons (column to the left), it is sent to specific routers (or intermediate nodes). Each router that receives it will broadcast the spike to all the neurons in its cluster. Once the spike arrives at a neuron, the address of the sender neuron will be compared to the one written in the neuron's memory, and if they match, the spike will trigger some action.

that has been done opens up a new path, an optimized placement for analog and mixed-signal multi-core processors is often neglected and relatively unexplored.
In this Thesis, we present a new approach for placement and routing of SNN onto analog/mixed-signal multi-core processors. All neuromorphic systems have a limitation on the maximum connectivity they can offer, which depends on the amount of memory available to save the connectivity structure of the network. We can notice the difference in size on the neuromorphic chips presented in Table 2.1. One main characteristic of that is the fact that digital technology still employs a single memory block to store the connectivity among neurons. In contrast, in mixed-analog technology, every neuron has its own memory block. By focusing on reducing the memory needed to save the connectivity in mixed-analog signal hardware, we provide a way to minimize the total chip die area.

"Sometimes science is more art than science, Morty. A lot of people don't get that."

Rick Sanchez, *Rick and Morty*

3. Brain-inspired routing

A software-hardware co-design approach is a cyclic process: the software depends on the hardware, and the hardware depends on the software. However, we can only discuss placement algorithms after introducing the routing scheme and the hardware design. Our hardware design (and routing scheme) was defined by considering the mixed-signal "emulation" approachBenjamin et al. (2014); Qiao et al. (2015); Moradi et al. (2018) and by keeping in mind the small-world network structure of our applications and the brain.

Finding trade-offs to optimize weight-matrix connectivity and routing memory structures in multi-core neuromorphic processors can significantly impact their total chip die area and the size of the networks they can implement.

Following the original neuromorphic engineering approach (Mead, 1990), we look at animal brains for inspiration and propose a brain-inspired strategy to perform this trade-off. Research on the effects of spatial constraints on brain connectivity can give us insights and a better understanding of the principles shaping neuronal organizations.

Our primary idea is to have a core fully connected in a hardwired way, representing groups of different populations. Each core could have a ratio between neuron' types (excitatory-inhibitory, following biology) already connected: all excitatory to all inhibitory and vice-versa.

We want to investigate how to connect neurons to allow an SNN with a high fan in/out. While maintaining the ability to provide a scalable system. The scalability of neuromorphic systems is mainly restricted by communication requirements (to root AE among neurons): bandwidth, latency, and memory requirements. Tipically, in neuromorphic chips, the memory area takes most of the chip size, thus we decided to focus on memory requirements.

The crossbar structure, introduced in Chapter 2, allows us to have many connections, with physical mapping of the network and processing direct in memory; however, we need to provide a practical route scheme to give rise to plastic and adaptive connections.

With this in mind, we propose a placement and routing scheme optimized for small-world networks that will minimize memory requirements in the routing tables while allowing flexibility for other network configurations.

The main characteristic we consider is the number of connections among neurons and how these connections become sparse with distance. As we find in the brain, the number of connection within regions is higher than between regions. We can see diferent brain regions as different cores in the neuromorphic hardware (see Figure 3.1 for a comparison). The connectivity inside a core would be the highest, and the number of connections decays across cores, given their physical distance.

Often, neuromorphic hardware has a fixed cost to connect two neurons: the price of a synapse is specified in terms of bit address, independently of their characteristics. We need to use CAMs or TCAM to store these bit addresses. The networks' fan-in and fan-out are limited by the available memory on hardware to store the connectivity. To scale up the number of connections allowed on a chip, we need to either provide more memory or reduce the cost of storing the connectivity. Increasing the chip memory leads to an increase in costs and area.

When we look at the small-world topologies, we notice that nearby neurons have the highest number of connections (dense, highly connected nearby clusters), and the connections get more sparse with distance. Our approach to designing a routing architecture is based on the idea that a connection's cost should consider the distance between the connected neurons, i.e., by using more or fewer bits to define connections depending on how often they are created. This is the same idea behind data compression used, for instance, in jpeg (Wallace, 1992), or morse code (Burns, 2004). In our approach, the memory needed to route a spike is inversely proportional to the distance between the connected neurons. Instead of considering all the connections with the same number

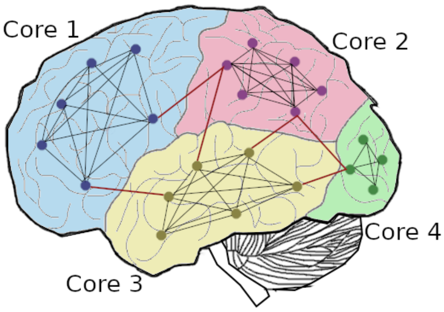


Figure 3.1.: Schematics of the small-world network connectivity in the brain, and how this became our inspiration for a new routing scheme. The number of connections inside a brain region is higher when compared to the number of connections between areas. Related brain areas are placed together and have more connections than unrelated areas. We can consider different brain regions as different cores. Cores that share many connections should be placed closer, and unrelated cores (with a small number of connections) should be placed far away.

of bits, we define the number of bits necessary for a connection based on their distance. The higher the distance between connected neurons, the more memory (i.e., bits) is needed to define a connection.

3.1. Distance matters

To reduce the memory requirements in this new routing, we reinforce the idea that distance plays a role in the connectivity, and mainly, it should be taken into consideration in the cost of creating connections on a chip.

For this, we introduce the concept of specificity. In densely connected clusters, neurons respond to all the activity of the neurons in the same population. In this case, there is no specificity: it does not matter the neuron sending the spike; all the neurons in the same population would respond to it[1]. The connection specificity increases with distance, inversely proportional to the number of connections. Since connections get more specific with distance, then more information is necessary; thus, the cost of creating a connection increase with how distant the connected neurons are. In biology, this is represented by the wiring cost. In neuromorphic, we define it as the number of bits needed to describe such a connection. Figure 3.2 shows an overall scheme of the specificity and the bits necessary to address the connections.

Our connection specificity is defined by the distance among neurons. We can implement this physical distance into our routing scheme in a hierarchical way. Depending on the level in the hierarchy we are (thus, the router level), we will need more or less bits to define connections. In our definition, when we move up in the router levels hierarchy, our specificity increases, i.e., we can be more specific regarding what subset of neurons is sending connections. With the increase in specificity more bits are necessary to address a spike, increasing the cost of creating a connection. At the highest router level, it would be possible to assign specific connections between two neurons, which leads to the identification of the complete address of the source neuron.

The fan-in per neuron in such a structure depends on the distance among cores and the number of cores at each distance, as explained in Figure 3.3.

Following the exponential drop-off of connection numbers with distance observed in biology, we assume that the number of connections required between cores depends on their distance.

[1] we consider all-to-all connected clusters, but we still can programmatically switch off some of the synapses to achieve other patterns of local connectivity (for instance, having 75% of the neurons on a core connected)

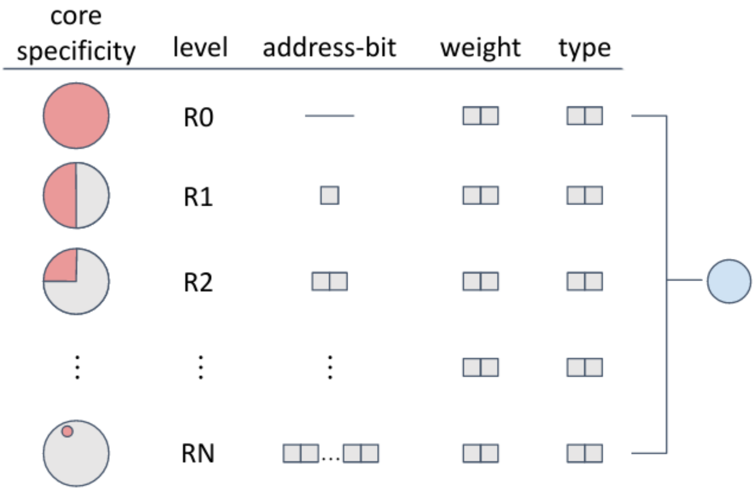


Figure 3.2.: Core specificity and the number of bits per neuron. Each circle represents a core, and the red portion of it shows the proportion that a destination synapse can listen to (can receive from) a source neuron. With increasing distance (and router level), we need more and more bits to define connections. The specificity of a core depends on the target neuron's "point-of-view". A neuron can receive connections from all the neurons in all the cores that can be reached through an $R0$ router, i.e., all the neurons in the same core. These connections are hardwired, with a single latch per core to (de)activate them. A neuron can receive connections from half of the neurons in all the cores that can be reached through an $R1$ router. These connections cost a single bit per neuron to listen to the right or left side of the cores. The number of possible connections is halved by every increase in router level, to the extent that we reach the specificity of a single neuron to far away cores. The number of bits to define a single-neuron connection depends on the number of neurons per core. Note that the total memory cost for every neuron needs to account for the bits used for weight and synapse's type ($GABA_A$, $GABA_B$, NMDA, AMPA, among others); Nevertheless, since those bits are constant, we will not consider them in calculating the number of bits necessary to create a connection.

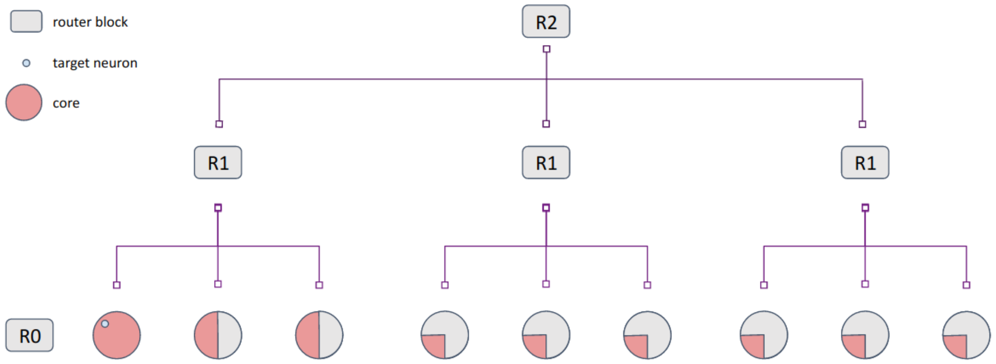


Figure 3.3.: Hierarchical fan-in. A target synapse of a neuron, represented as the blue dot in the left most core, has a fixed number of possible connections. The number of connections it can receive depends on the distance/router and the number of cores at that level. Every node in the tree represents a core. *Red areas* are the core portion that can target the marked neuron (depicted in blue). In this example, assuming three router levels ($R0$, $R1$, and $R2$), nine cores, and four neurons per core (a total of 36 neurons in the chip), the target neuron can receive spikes from all the neurons in the same core (4 neurons); half of the neurons in the cores reached through $R1$ $(2*2)$; and a fourth of the neurons in the cores reached through $R2$ $(6*1)$.

We associate physical distance with the levels in the router hierarchy: e.g., all cores that can be reached via an $R1$ router level are at a distance 1. Each neuron in a core can receive inputs from all the neurons inside the same core (all the neurons are seen as one single block at $R0$); from one of the halves of all the cores reached through an $R1$ router; from a fourth of all the neurons in each core reached through an $R2$; and so on[2].

As the distance between neurons and cores increases, fewer connections are made; thus, there is no need to allow connectivity between all neurons in different cores. This allows us to reduce the connectivity address space and thus reduce the overall memory required to specify each neuron's source and target population address.

Thus, considering n neurons per core, the fan-in of a single neuron can be defined from the router levels. From R0, a target neuron can receive spikes from n neurons (all neurons inside the core), from R1, it can receive spikes from $\frac{n}{2}$, from R2, $\frac{n}{4}$, from R3, $\frac{n}{8}$, and so on. Or, more formally:

$$\text{Fan in per neuron} = n + \Big(\sum_{i=1}^{r} \frac{n}{i*2}\Big) \times N_r \tag{3.1}$$

Where r is the number of router levels, and N_r is the number of cores that can be reached through that router.

3.2. Synapse types

To create the connectivity on hardware it is necessary to store the information about source and destination neurons in memory. In our approach, the distance between two connected neurons is an essential factor in determining the cost of the connection. And, as shown in Figure 3.2, we have a different number of bits needed to create a connection between neurons, given the distance between them. At first glance, TCAMs seem more appropriate for our routing scheme since we could have the same number of bits for all connections and use the "don't care" bit to give

[2] In our scheme, we also provide a few rows of "full address bits" to specify connections that could break the network structure we are focused on and make the chip more general.

rise to the specificity. However, TCAMs are more costly in terms of area and power than CAMs cells (Liu, 2002; Noda et al., 2005). With this in mind, in this work, we use CAM for our routing with variable lenght addresses.

In our hierarchical architecture, we need to define new synapse types[3]. Mainly, two types are vital to be differentiated: the local ones, i.e., synapses created inside a core, which we will call *local recurrent synapses*, and the synapses between cores, which we will call *non-local*.

3.2.1. Local recurrent synapses

The local recurrent synapses are the connections formed inside the core using $R0$ routers. In our architecture, the local connections can be either all-to-all connected or not connected at all. This imposes hard constraints in terms of population size in our network. Since all cores have the same size, thus the same number of neurons, and with only an all-to-all connectivity option, we would have to assume all the populations in the network have the same amount of neurons.

Often, this is not the case. To overcome this limitation, a latch per neuron is included. We can create variation in population connectivity with software programmability, where we can shut off some connections, for instance.

3.2.2. Non-local synapses

Any connection between neurons from different cores is defined as an external synapse. This means the connection has to go through, at least, the $R1$ router.

The $R1$ routers connect a fixed number of cores. Upper-level (>$R2$) routers connect a fixed number of lower-level routers.

They manage inter-core communication and long-distance communication. Each connection has a different cost associated with it, given the router level it needs to go through.

Also, we have off-chip synapses, which can be used to inject input signals from outside the chip. A few rows of full address bits are available for those connections.

Learning synapses

Biological networks adapt and learn continuously. In a wide variety of species, from insects to humans, this learning is achieved by changing the synapses' strength (or weights). One classical way to change this strength depends on the firing order between a pre- and post-synaptic neuron (Caporale and Dan, 2008).

In neuromorphic hardware, this can be implemented using a crossbar or any other memory-structured architecture. Crossbars' driving signals are applied to the axon and dendrite to update the weight stored at their intersection. In other memory-designed architectures, some bits of memory store the weights that can be updated given the proximity of an input spike and an output spike, which can be stored in a buffer or used as another signal.

Often the circuits that allow learning mechanisms are designed separated and are implemented in a way that can be disabled. In this work, we do not consider learning circuits and assume our networks will be placed after training, being used for inference.

3.3. Multi-core hierarchical routing mechanism

We propose a new multi-core hierarchical routing scheme that exploits the SNN architecture to reduce memory consumption and address the connectivities. In our hierarchical routing scheme, we have different levels of routers. And the maximum router level needed is dependent on the number of neurons per core. Each router is responsible for the local traffic, deciding if the spikes (or events) should be broadcasted down, sent to other routers at the same level, or to a router in the upper hierarchy.

[3] Please, bear in mind these types have nothing to do with the classical synapse types as in excitatory or inhibitory.

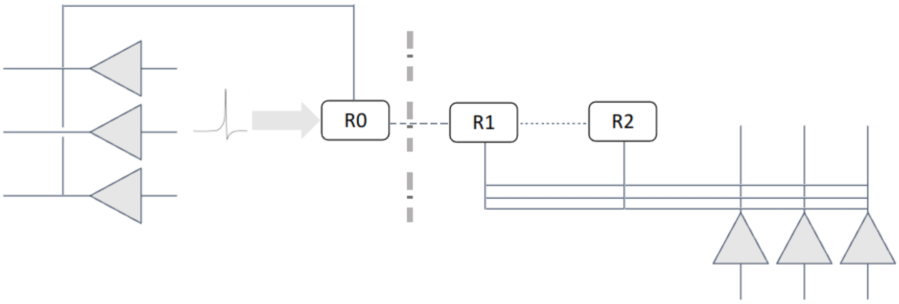


Figure 3.4.: Hierarchical routing with logic inside routers and a combination of source-address and destination-address schemes. When a source neuron generates a spike, this spike will contain the source neuron address and the information about the distance that the spike needs to reach. The spike arrives at the $R0$ (local core router), and it will be broadcasted inside the core (without self-connection). In each router level, including the $R0$ router, the router identifies if the spike needs to be sent to a farther distance and, if so, sends it to the next router level. Every router also sends the spike to the neurons connected to it. In this figure, we have a representation of a small part of the chip; not all neurons will be connected to the same $R1$ routers, for instance, so not all spikes will arrive at all neurons in the chip. When a target synapse receives the spike, it gets the information about the source neuron address and the distance the spike is coming from; thus, it can identify how many bits are necessary to make a matched comparison to accept the spike.

The source neuron just needs to know how far its target neuron is. This way, when a neuron fires, the spike is sent to its first router, $R0$. Every router contains instructions that, given the distance the spike needs to travel, can decide if the spike will be broadcasted down, sent to an upper-level router, or another router in the same hierarchy level (see Figure 3.4 for a schematics).

The information sent in the event also contains the full address of the source neuron. However, when a neuron receives an input spike, it only checks part of the address given the router level that the spike came through. Any spike sent by an $R0$ router is not even evaluated; the neuron accepts the spike and reacts to it. Spikes sent by an $R1$ router have the first bit checked with the information in the target synapse neuron memory for $R1$ router: if the source neuron is located in the expected part of the core, then the target neuron will accept and react to the spike; otherwise, nothing happens. Similar checks are applied for higher-level routers ($> R2$): for every router level added, one more bit needs to be verified.

These connections have a constrained address space, i.e., since there is an upper bound on the router to which a spike can be sent, it is not necessary to consider all of the neurons on the whole chip. In a model going up to the $R2$ level, each neuron needs memory to store seven bits in total, allowing a fan-in from up to half of the neurons in the cores that can be reached through the $R1$ level plus a fourth of the neurons in each of the cores that can be reached through $R2$. To support this reduction in address space, we compute the routing distance by combining the use of computing logic and memory in each router module and update the distance information in the spike packet as it traverses the router, thus reducing the address space to the bare minimum needed by the local cluster.

Zhe Su currently develops all the designs of this new chip and routing architecture. Figure 3.5 shows the hierarchical routing scheme designed to support small-world network connectivity with on-chip memory.

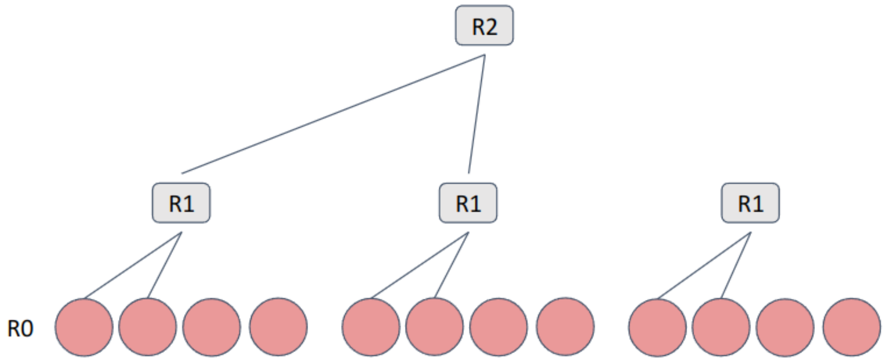


Figure 3.5.: Hierarchical routing NoC architecture being designed by Zhe Su. Each red circle represents a core. This hierarchical tree architecture is optimized for small-world network topology. This structure reduces the number of routing hops and allows a large fan-out multicast, thus reducing the power consumption of routing.

3.4. Summary

The routing scheme defines how neurons can communicate and the way memory needs to be distributed to allow the connectivity. In our hierarchical brain-inspired routing scheme, we have densely connected clusters, without neuron specificity inside cores. Following the exponential drop-off of connection numbers with distance observed in biology, the routing scheme considers that the number of connections required between cores depends on the distance between them. We represent the physical distance with the levels in the router hierarchy: e.g., all cores that can be reached via an $R1$ router level are at a distance 1, via $R2$ routers are at distance 2, and so on.
There is a clear relation between router levels, number of connections, and distance between cores. As the distance between neurons and cores increases, fewer connections are made; thus, there is no need to provide resources to create connections between all the neurons in different cores. This allows us to reduce the connectivity address space and thus reduce the overall memory required to specify the source population address for each neuron. In our hierarchical routing scheme, high-level routers manage connections between the immediately low-level routers. The router level just above the cores manages the connections between a fixed set of cores. The trade-off implemented is to add logic to the routers and combine multicast source address routing with destination address routing. Specifically, by adopting small-world type network size/connectivity, we can implement trade-offs that minimize memory requirements while still enabling the design of SNN architectures that can solve a wide range of relevant problems, i.e., the types of sensory-motor processing problems that animals solve in the real world. We can minimize memory requirements by reducing the address space (i.e., the number of bits and hence chip area) required to map the (many) connections between nearby neurons and allocate more bits for larger address space domains used by the sparse long-range connections.

"Programming is more than an important practical art. It is also a gigantic undertaking in the foundations of knowledge."

Grace Hopper

4. Brain-inspired placement algorithm

The routing scheme defined in Chapter 3 determines the source and target memory structures based on a *relative* distance between neurons and cores. The placement algorithm needs to place neurons and synapses of a user-defined neural network into neuron circuits and cores located in *absolute* space coordinates while adhering to the hardware's specificity and distance-based connectivity constraints. For every neuron in the SNN, a specific location on the neuromorphic chip has to be defined. The quality of the neuron placement is essential to keep routing the spikes efficiently. The new hierarchical routing scheme, although providing a way to minimize memory resources, defines constraints that impact the placement of a network on the chip. Thus, offering an automatic placement increases the usability of the chip.

4.1. A canonical network

To define our placement algorithm, we started by defining a *canonical* network. This is the network that follows our hardware constraints and leads to optimal use of the hardware resources. We took as inspiration for the canonical network examples of WTA networks introduced in Chapter 1, as shown in Fig. 4.1, with small-world connectivity matrices.

Our canonical network has neuron populations with the same number of neurons as in our neuromorphic core. Each population is then all-to-all connected, and the number of connections between populations drops off with the distance between the cores, as depicted in Fig. 4.2.

4.2. The placement algorithm

We propose a new brain-inspired heuristic to place the network on this new hardware. In Chapter 3 we introduced our hierarchical routing scheme designed to support small-world network connectivity with on-chip memory.

The routing scheme defines some rules for our placement. The lack of specificity inside cores and its densely connected set of neurons indicates we need to find densely connected clusters in our SNN to be mapped to cores. The exponential decay in connections between router levels is associated with physical distance, e.g., all cores that can be reached via an $R1$ router level are at a distance 1. This information indicates how we should place cores regarding each other. There is a clear relation between router levels, number of connections, and distance between cores.

The routing scheme gives us some restrictions about the hardware we need to incorporate in our placement. We have highly connected clusters and a dependency between core distance and the number of connections. With all of this, to place a network, we follow three main steps:

- neuron placement: first, we place neurons in cores given the connections among them; we look for densely connected clusters in the network.

- distance calculation: second, we create a map of distances among the cores created; the number of connections between cores defines our distances.

- synapse assignment: lastly, we place the connections among neurons; we define a router path for each connection.

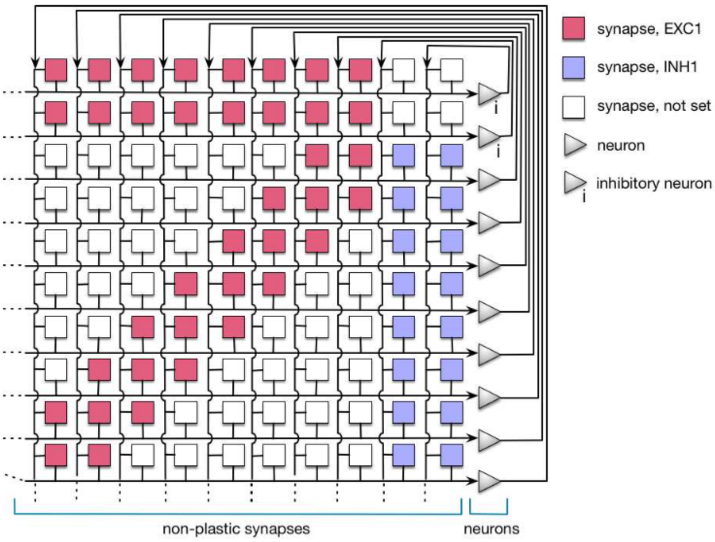


Figure 4.1.: Example of a WTA network implemented on a neuromorphic processor (from (Indiveri and Sandamirskaya, 2019)). Blue blocks represent inhibitory synapses with negative weights, and red blocks represent excitatory synapses with positive weights.

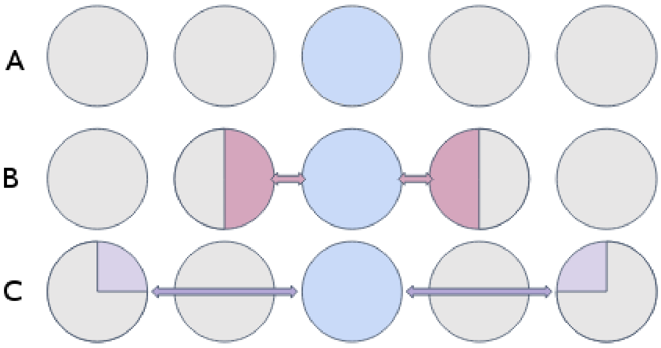


Figure 4.2.: Canonical networks are generated to match a hypothetical neuromorphic processor. Row A shows five populations. We create populations with the same number of neurons as in a core, all-to-all connected. For simplicity, we place our populations on a 1d-line defining distances (and thus the connections between populations). In B, we assign connections between cores at a distance of 1. These cores will share connections with half the neurons in a nearby core. In C, we assign connections between cores at a distance of 2. They share connections with a fourth of the number of neurons in a core at that distance. This process is repeated until there are no more cores or they are so far away that no connection is created. Only the procedure for the central core is shown in the figure, but this process is applied to all cores.

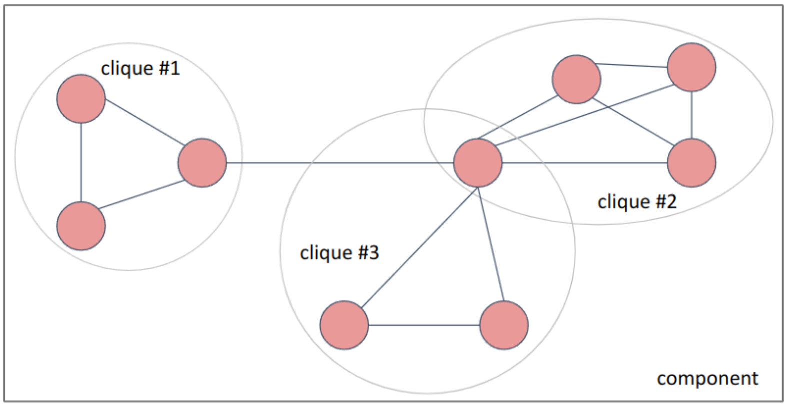


Figure 4.3.: Diference between a clique and a component. In a clique, all the nodes are directly connected to each other. In a component, all the nodes are directly or indirectly connected to each other.

4.2.1. Neuron placement

The placement of neurons starts by looking at the neurons as "receivers", i.e., considering the connections arriving at a neuron. For this, we look at the adjacency matrix only with the incoming connections to a neuron.

We group neurons that share the largest number of common senders. And because we have an architecture where inside a core, the neurons are all-to-all connected, we search for cliques of the network. Cliques are a particular case of common senders, where the whole group receives spikes from every other neuron. We use the Bron-Kerbosch algorithm with pivoting to define our cliques (Bron and Kerbosch, 1973; Cazals and Karande, 2008). A neuron, however, can be part of one or more cliques. Since the same neuron can not be placed in two different cores in the hardware, after generating cliques, we verify that every neuron is, at most, in one clique. If a neuron is part of two or more cliques, we use a heuristic to keep the neuron in a single clique. Our heuristic verifies if we can keep the neuron in a core without exceeding the maximum number of neurons in the core. We then place the neuron in the biggest clique that is within the core capacity. Each clique will be placed in a different core. The routing of spikings in these cliques needs virtually no memory since the connections are hardwired. After placing cliques, if any neuron is left, i.e., if a neuron is not part of any clique, we keep forming clusters considering the largest number of common senders. Each cluster will be added to a different core.

Algorithm 1 describes more formally how we place neurons in cores given the connections among them.

4.2.2. Distance calculation

With all neurons separated by cores, we define distances among the cores, to be able to place specific connections. The number of connections between two cores defines the distance between them. Since we follow the biological concept that the number of connections decays with distance, we define the distance between cores i and j as:

$$dist(i,j) = \left\lfloor \frac{n}{e(i,j)} \right\rfloor + 1 \tag{4.1}$$

where n is the number of neurons per core, and $e(i,j)$ is the number of connections core i is receiving from core j. Note that our distance metric can be non-symmetric, i.e., the distance from core i to j might not be the same as from core j to i.

[1] a clique is a group of nodes that are all connected to each other, see 4.3

Algorithm 1 Neuron placement

```
Require: Graph $G = (V, E)$ with incoming connections and $coreSize$ with maximum number of
  neurons that can be added to a core
  run Bron-Kerbosch algorithm to find all maximal cliques[1]in the graph.
  mark all neurons that are in in more than one clique
  for cliques: biggest to smallest do
    if neuron is in more than one clique then
      if $cliqueSize <= coreSize$ then
        keep neuron in the clique, remove from others
      else
        remove neuron from the clique
  $numCores = 0$
  $currentCore = -1$
  initialize a core
  for cliques do
    $cliqueSize$ = number of neurons in the clique
    $coresNeeded = cliqueSize/coreSize$
    $numCores+ = coresNeeded$
    initialize $coreNeeded$ cores
    $currentCore++$
    for neurons do
      if $currentCore$ has space then
        add neuron to $currentCore$
      else
        $currentCore++$
        add neuron to $currentCore$
  if there are neurons not placed (not part of any clique) then
    find cluster among the sender neurons
    for each cluster do
      $coresNeeded = cliqueSize/coreSize$
      add target neurons to a new core while they fit a core and instantiate more cores if needed
      $numCores+ = coresNeeded$
  if there are neurons not placed then
    $coresNeeded$ = number of neurons not placed $/coreSize$
    add neurons to a new core(s)
    $numCores+ = coresNeeded$
```

Algorithm 2 shows how we create a map of distances following our design choice.

Algorithm 2 Distance calculation

```
Require: Graph $G = (V, E)$, where $V$ are the cores, and the weight in every edge is the number of
    connections among them.
    $dist[i][j]$ initialized with $-1$
    $n$ is the number of neurons per core
    $e(i, j)$ is the number of connections core $i$ is receiving from core $j$.
    for each core i do
        for each core j do
            if $i == j$ then
                $dist[i][j] = 0$
            else
                $dist[i][j] = math.floor((n/e(i, j))) + 1$
```

Given that the distance calculation step can generate non-symmetric distances between cores, but the physical placement doesn't support that, we also run a verification step to make the distances between cores symmetric.

If two cores have non-symmetric distances, we can make them symmetric by using the closest or farthest distance. We can choose what distance will be used given the number of inconsistencies we generate while assigning the routing path.

To illustrate an approach in using the farther distance, consider a core with 32 neurons. At $R1$ level, the maximum fan-in of a neuron is 16 neurons per core. At $R2$ the maximum is 8. Now, suppose core A sends 9 connections to core B. The distance between A and B will be 1, and they need to communicate from $R1$ router level. If core B sends 1 connection to core A, we will assign the distance between core B to A as 5. By using the farther distance to make the distance between cores symmetric, we have 8 connections from core A to B marked as inconsistent (since with a farther distance, we can't deal with such a high number of connections). If we decided on the closest distance, thus defining the distance from B to A as 1, we could place all the connections. However, we need to check that all other neurons in the same half as the sender neuron (i.e., the additional 15 neurons in this half core) are not sending connections. Otherwise, the neuron in core A would receive more spikes than stipulated. Our primary criterion for placing neurons into cores considers that neurons are all-to-all connected; by default, we don't assume a configuration where only selected neurons would spike. But it can happen if populations have less neurons that fit in the portion of the core sending the connection.

In case one core is sending connections to another but without receiving connections back, we add them even further away, to be able to have a more specific target connectivity. We need to make the number of connections being sent fit the next router level distance.

Inconsistency An inconsistency can occur when neurons at a core have to listen to different areas from cores at the same router level or when connections can not be placed. If a neuron placed in a target core is receiving connections from sender neurons in the left half of a core reached through an $R1$ router, then the neuron in the target core can only listen to the left side of all cores reached through the $R1$ router. Figure 4.4 shows an example of an inconsistency.

4.2.3. Synapse assignment

The synapse assignment is the solution to the routing problem: for every synaptic connection, we need to define what set of hardware lines will be used to connect the output of a presynaptic neuron to the input of a post-synaptic neuron, or in other words, the path the spike will traverse. To finally place the connections, we now consider the neurons as "senders", meaning that we want to form groups of neurons inside the same core based on where they send their spikes. For each core, we identify what group of neurons has the same post-synaptic neuron. We need to allocate

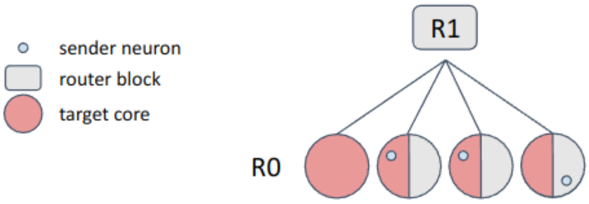


Figure 4.4.: In our scheme, a target core can receive spikes from the same subset of neurons in all the cores reached through the same level router. At the $R1$ level, each neuron can listen to either the left or right side of the other cores. For instance, the target core in red can receive connections from neurons placed in the red area (reached through $R1$ router). If a neuron needs to listen to two different areas, as depicted in the image, this generates an inconsistency.

these neurons to different "slices" inside the same core (see Figure 3.3 for a better understanding of slices and their specificity).

We start this process by allocating the neuron into slices of cores that are nearby. This way, we guarantee the placement of the highest amount of connections. The idea is to force the inconsistencies to be generated as distantly as possible. Since the number of connections decays with distance, this let us generate a minimum number of inconsistencies.

The inconsistencies are marked in the connections, not in the neurons. If a neuron cannot be placed in a specific slice, all its connections are flagged as inconsistent. As described in Section 3.1, we want to provide some rows of "full address bits" in our design. These rows can be used to place some of the flagged connections.

After a first loop placing the connections, we can go over all the inconsistencies generated. For every connection marked as inconsistent, we check all the connections the sender neuron has. If the number of flagged connections is higher than the number of placed connections, we check if moving this neuron to a new core would reduce the number of flagged connections. If so, we change the neuron position; otherwise, we try to fit the inconsistencies using the "full-address bits" rows.

Algorithm 3 describes the placement of the connections among neurons.

4.3. A placement example

Let's consider a simple network that follows our hardware design. Figure 4.5 shows the distribution of connectivities. This network contains 16 neurons grouped in four clusters of four neurons each. Each cluster is connected to the neighboring clusters, with decay in the number of connections (as seen in the right side of Figure 4.5). Following our approach, each cluster would be placed in a different core, which would lead to the use of four cores.

Let's have a closer look at the connections among the populations. To make easier the idea of distances, let's consider that the four populations are placed on a 1d line, as described in Figure 4.6. The most distant connections determine the router level we need to place the network following our hierarchical approach. In this example, the maximum distance for connections is 2. Note that there are cores apart by distance 3; however, since there is no connection among them, we need the router up to distance 2.

This seems evident by looking at the network, but to determine the level of router needed, we need to count the number of connections among the cores. Figure 4.7 shows how the connections are distributed among cores more visually.

By definition, all cores receiving more than half of their connections compared to the number of neurons in a core (in a pairwise manner) are at a distance 1, between a fourth and half, at a distance 2, and so on. We calculate the distance using Equation 4.1. As an example, core 0 is

Algorithm 3 Synapse assignment

```
Require: Graph $G = (V,E)$ with outgoing connections and matrix $dist[i][j]$ of the distances between cores
  sort all pairwise cores according to the distances between them (from closest to farthest).
  for each pair in the sorted list do
    if $dist[i][j] <= 0$ then
      continue
    for each neuron in the receiving core do
      gather all neurons in the sending core that are sending connections to the target neuron
      verify if neurons fit in the slice size (Algorithm 5)
      if neurons sending connection were not placed yet then
        find a slice available
        add neurons sending connection to the slice
        mark target synapse to listen to the slice
      else
        if all neurons were already placed and are in the same slice then
          evaluate target neuron (Algorithm 6)
        else
          if not all neurons were placed then
            if neurons placed are in the same slice then
              add the unplaced neurons to the slice
              evaluate target neuron (Algorithm 6)
            if some neurons were placed but not on the same slice then
              if target neuron is not listening to any area then
                identify slice with majority of connections
                target neuron listen to that slice
                add unplaced neurons to that slice
                flag connections from neurons in other slices as inconsistent
              else
                add unplaced neuron to the slice target neuron is listening to
                flag connection from neuron in other slices as inconsistent
  evaluate inconsistencies (Algorithm 4)
```

Algorithm 4 Synapse assignment - evaluation of inconsistencies

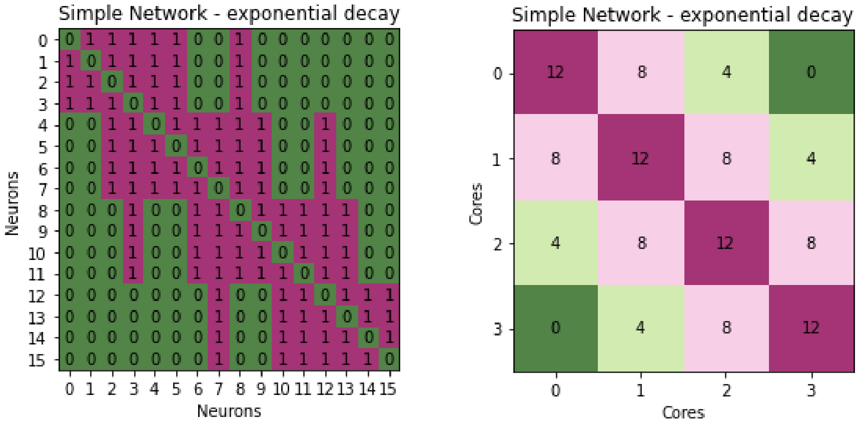
```
Require: list of flagged connections
  for every flagged connection do
    count the number of flagged connections for the sender neuron
    if number flagged connection > not flagged connection then
      move neuron to a new core
      recount number of flagged connections
      if number flagged connections decreased then
        keep neuron in the new location
      else
        move neuron back to original position
  for every flagged connection do
    if target neuron has full adress rows available then
      place connection
  if number of flagged connections is > 0 then
    notify the network couldnt be placed completely
```

Algorithm 5 Synapse assignment - verification of slice size

```
Require: set of neurons sending connections and size of the slice
  if number of neurons sending connection > slice size then
    sort neurons by number of sending connections
    while number of neurons sending connection > slice size do
      get neuron with small number of connections
      flag its connections as inconsistent
      remove neuron from the sending connection group
```

Algorithm 6 Synapse assignment - evaluation of target neuron

```
Require: synapse information on target neuron and position of source neurons
  if target neuron is not listening to any area then
    target neuron listens to the area of the source neurons
  else
    if target neuron is listening to another area then
      flag connections as inconsistent
```



Simple Network - exponential decay

Neurons	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	1	1	1	1	0	0	1	0	0	0	0	0	0	0
1	1	0	1	1	1	1	0	0	1	0	0	0	0	0	0	0
2	1	1	0	1	1	1	0	0	1	0	0	0	0	0	0	0
3	1	1	1	0	1	1	0	0	1	0	0	0	0	0	0	0
4	0	0	1	1	0	1	1	1	1	1	0	0	1	0	0	0
5	0	0	1	1	1	0	1	1	1	1	0	0	1	0	0	0
6	0	0	1	1	1	1	0	1	1	1	0	0	1	0	0	0
7	0	0	1	1	1	1	1	0	1	1	0	0	1	0	0	0
8	0	0	0	1	0	0	1	1	0	1	1	1	1	1	0	0
9	0	0	0	1	0	0	1	1	1	0	1	1	1	1	0	0
10	0	0	0	1	0	0	1	1	1	1	0	1	1	1	0	0
11	0	0	0	1	0	0	1	1	1	1	1	0	1	1	0	0
12	0	0	0	0	0	0	0	1	0	0	1	1	0	1	1	1
13	0	0	0	0	0	0	0	1	0	0	1	1	1	0	1	1
14	0	0	0	0	0	0	0	1	0	0	1	1	1	1	0	1
15	0	0	0	0	0	0	0	1	0	0	1	1	1	1	1	0

Simple Network - exponential decay

Cores	0	1	2	3
0	12	8	4	0
1	8	12	8	4
2	4	8	12	8
3	0	4	8	12

Figure 4.5.: A simple network with 16 neurons and the decay of connectivity given the distance among neurons.

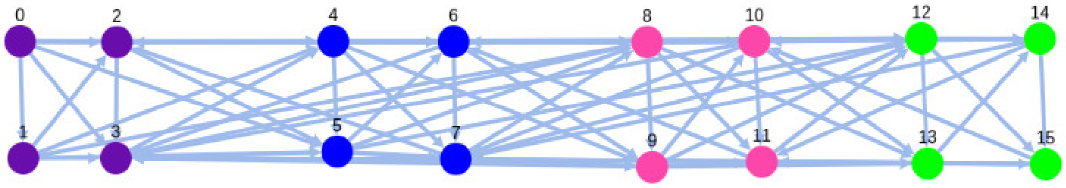


Figure 4.6.: Simple network and its connections among populations. Clusters were placed on a 1d line. Purple neurons are at a distance of one from blue neurons, two from pink, and three from green neurons. Blue neurons are at a distance of one from purple and pink neurons and a distance of two from green neurons. Pink neurons are at a distance of one from blue and green neurons and a distance of two from purple neurons. Green neurons are at a distance of one from pink neurons, two from blue neurons, and three from purple neurons. **Note that the distances among cores is symmetric.**

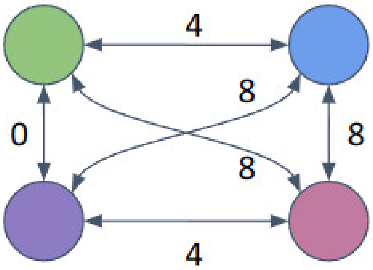
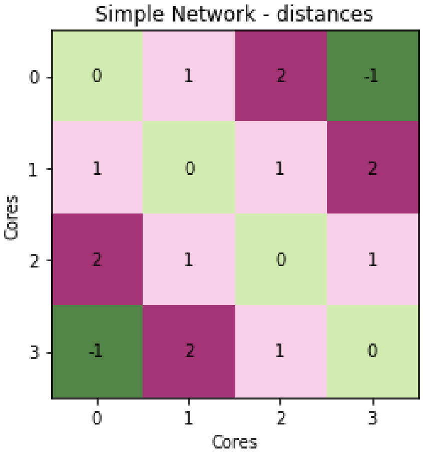


Figure 4.7.: Connectivity among cores. Each square represents a core with a set of neurons. The values on the edges are the number of connections each core is receiving. The connections here are seen on a more general level, i.e., it doesn't matter which neuron will receive the connection; we are just interested in the connections arriving at the core. floor



	0	1	2	3
0	0	1	2	-1
1	1	0	1	2
2	2	1	0	1
3	-1	2	1	0

Figure 4.8.: Map of the distance between cores. The distance gives us information about what router level we need to use. Distance zero means a core is talking to itself. Distance -1 means there is no definition of distance between the cores.

receiving eight connections from core 1, thus $d(0,1) = \lfloor \frac{4}{8} \rfloor + 1 = 1$ With this, we can create a map of distances between all cores, as seen in Figure 4.8.

With the distances defined, we know the maximum router level needed to place the connections. To place our example network, after defining the number of cores and distances among them, we loop over all neurons, starting with nearby cores. In this example, we will place connections between all the cores at a distance of one, then a distance of two, and so on.

Now we look at the connections that go across cores. We start with the connection from core 1 to core 0. Neuron 0 is receiving connections from neurons $4,5$. For core 1, we can mark neurons $4,5$ to the right and $6,7$ to the left. Neuron 0 writes 1 to the 1-bit $R1$ level to receive spikes from the right side of $R1$ cores. Neurons $1,2$, and 3 receive the same connections as neuron 0, so the distribution of connections is the same.

From core 0 to core 1: neuron 4 receives connections from neurons 2 and 3. For core 0, we mark neurons $2,3$ to the right and $0,1$ to the left. We do the same with neurons $5,6$, and 7, keeping the results of neuron 4. Neurons in core 1 receive spikes from the right side of $R1$ cores.

From core 2 to core 1: neuron 4 receives connections from neurons 8 and 9. For core 2, we mark neurons $8,9$ to the right and $10,11$ to the left. We do the same with neurons $5,6$, and 7, keeping the results of neuron 4. In core one, neurons receive spikes from the right side of $R1$ cores.

From core 1 to core 2: neuron 8 receives connections from neurons 6 and 7. Neurons 6 and 7 are already assigned to the same side, so we keep it this way, and neurons on core two are marked to listen to the left side of $R1$ cores. We keep the same results for neurons $9, 10$, and 11.
From core 3 to core 2: neuron 8 receives connections from neurons 12 and 13. Since no neuron was placed yet in core 3, but neurons in core 2 are already listening to the left side of $R1$ cores, we mark neurons 12 and 13 as to the left, and 14 and 15 to the right side. The same results are kept for neurons $9, 10$, and 11.
From core 2 to core 3: neuron 12 receives connections from neurons 10 and 11. Since neurons 10 and 11 are on the same side already (left), neurons in core 3 are marked to listen to spikes from the left side of $R1$ cores. The same results are kept for neurons $13, 14$, and 15.
Now we can place connections at a distance 2. From core 0, only neuron 8 is sending $R2$ connection. Neuron 8 was placed on the top-right corner, and neurons in core 1 can listen to the top-right area of $R2$ cores.
Core 1 receives a connection from neuron 12 at a distance 2. Neuron 12 was placed in the top-right corner; thus neurons in core one can accept spikes from the top-right area of $R2$ cores.
Core 2 receives a connection from neuron 3 in core 0. Neuron 3 was placed on the bottom-right corner of core 0; thus neurons in core two can listen to spikes from the bottom-right area of $R2$ cores.
Core 3 receives a connection from neuron 7 in core 1. Neuron 7 was placed at the bottom-left corner of core 1; thus neurons in core three can listen to spikes from the bottom-left area of $R2$ cores.
All connections were placed, and no inconsistency was found.

4.4. Summary

The placement algorithm defines how neurons and synapses can be allocated on hardware. We designed our hierarchical brain-inspired placement algorithm based on the hardware constraints imposed by the routing architecture. The physical relationship between cores and routers limits the number of connections that neurons in different cores can share. Our algorithm follows three steps: neuron placement, distance calculation, and synapse assignment. The neuron placement is responsible for allocating the neurons of a user-defined neural network into the hardware cores. This is done by finding groups of neurons that are all-to-all connected. After defining the position of neurons into cores, we calculate the router level we need to allow the placement of the existing connections. The router level is directly related to the number of connections two cores can share and their physical distance. The third step is the definition of synapse allocation. The synapse allocation can be seen as an arrangement of neurons inside each core to avoid or reduce the number of inconsistencies. An inconsistency is found when a synapse needs to receive inputs from neurons in different slices of a core or when the neuron does not have synapses available to add a connection. Our algorithm was designed to place canonical networks based on WTA architecture, which fit our hardware perfectly.

"You asked us a question and she knows the answer! Why ask if you don't want to be told?"

Ron Weasley, *Prisoner of Azkaban*, *Harry Potter*

5. Results

The placement algorithm described in Chapter 4 is one of the results of this Thesis. We not only provide a framework to place a network into the hardware structure, but we can also use it to analyze the hardware configuration parameters. Our algorithm was created based on the canonical network structure that matches our hypothetical neuromorphic processor, described in Section 4.1. As such, it succeeds in placing it optimally[1].

Here, we provide some memory use analysis and a cost function that can be used to calculate memory and area while designing new chips. Moreover, although the networks we focus on follow a similar topology, we expect real network applications to differ from our hardware structure. For instance, the networks we find in the brain or neuromorphic applications do not have all the populations with the same number of neurons. To validate our placement algorithm, we tested it with variations of our canonical network. Since the placement is not a trivial problem, the solution for generic networks is hard to assess. We don't know the Ground Truth (GT) for a network with different structures, such as randomly connected. However, by using a network that fully matches the hardware structure considered, we can define a GT placement and compare the placement of variations of the network to it.

5.1. Memory use analysis

While designing a new chip, parameters are often decided by an arbitrary design decision: number of neurons per core, number of cores, number of synapses per neuron, etc. All of these parameters incur a cost for chip fabrication. Deciding what parameters can still provide the necessary functionalities while reducing costs is extremely valuable when prototyping a chip.

To support the memory reduction assumption from our method, we present an analysis by proposing a function that, given some hardware configurations, can estimate the cost in terms of memory bits per neuron and chip area.

We present the calculation for our architecture and show how we calculate the costs for already known neuromorphic processors, namely, DYNAP-SE and TrueNorth.

5.1.1. Defining the cost function

In the following, we use A and R with various subscripts for areas and N with the same subscripts for numbers, indicated by the subscripts per next higher unit. The subscripts are C for core, N for neuron, S for synapse, and $\oplus$ for soma. Thus A_C is the area of a core, N_N is the number of neurons per core, N_C is the number of cores per chip, etc.

Thus, in a general form, the area of a chip is given by the area of all cores plus the area taken by the router:

$$A_{chip} = N_C A_C + R_{chip} \tag{5.1}$$

Where R_{chip} is the area for the *R*outing overhead at the chip level, described later in Subsection **Routing overheads**.

Similarly, the area of a core is given by the area of all the neurons in the core plus the area taken by the router at the core level. Expanding A_C gives

[1] By optimally, we mean the algorithm can find the neuromorphic processor structure that originated the network, thus not wasting neurons or cores.

$$A_{chip} = N_C(N_N A_N + R_C) + R_{chip} \tag{5.2}$$

R_C is the routing overhead at the core level.
The area of a neuron, A_N, is the area of the soma, plus the sum of the areas of all the synapses, plus the routing overhead within the neuron, R_N

$$A_N = A_{\oplus} + \sum_{t=1}^{T} N_{S_t} A_{S_t} + R_N \tag{5.3}$$

where there are T synapse types.
Substituting into (5.2) gives

$$A_{chip} = N_C(N_N(A_{\oplus} + \sum_{t=1}^{T} N_{S_t} A_{S_t} + R_N) + R_C) + R_{chip} \tag{5.4}$$

Synapse area

The area occupied by some synapse types may depend on factors such as the fan-outs of the routers and if they are learning synapses, for instance.
Generally, we can separate two main categories of synapses in any routing scheme: those called *Learning synapses* which participate in the learning process and will be denoted with the subscript S_L; and those called *Fixed synapses* which do not participate in the learning process and will be denoted with the subscript S_F. With these two synapse categories, equation (5.3) for the area of a neuron becomes

$$A_N = A_{\oplus} + N_{S_L} A_{S_L} \sum_{t=1}^{T} N_{S_t} A_{S_t} + N_{S_F} A_{S_F} \sum_{t=1}^{T} N_{S_t} A_{S_t} + R_N \tag{5.5}$$

In all of the above, the 'routing overhead' area terms R_* have not been expanded. Note that each of these terms should include not just the area required for active router elements but also the area necessary for de-multiplexers, multiplexers, etc. and also simple wiring attributable to the given level in the chip–core–neuron hierarchy. In practice, these terms will depend, in general, on the number of cores, neurons, and synapses:

$$\begin{aligned}
R_{chip} &= r_{chip}(N_C, N_N, N_{S_1}, \ldots, N_{S_T}) \\
R_C &= r_C(N_C, N_N, N_{S_1}, \ldots, N_{S_T}) \\
R_N &= r_N(N_C, N_N, N_{S_1}, \ldots, N_{S_T})
\end{aligned}$$

so we'll have to define the functions r_{chip}, r_C and r_N. These functions will be specific for each architecture used.

In a hierarchical architecture, R_{chip} is going to be composed of the area required for all the levels of router blocks R0, R1, R2, ... (and any other router levels needed)[2] in the router tree plus the area required for de-multiplexers, multiplexers, etc.:

$$R_{chip} = N_{R0} A_{R0} + N_{R \geqslant 1} A_{R \geqslant 1} + A_{dmux} + A_{mux} + A_{etc}$$

[2] If the hardware contains no hierarchical structure for the router levels, all the terms related with $R \geqslant 1$ can easily be ignored, simplifying the costs to the $R0$ term.

Assuming that A_{dmux}, A_{mux} and A_{etc} are going to be constant and independent of N_C, N_N, etc., we can replace the sum of these terms with α, and expand the N_{R*}'s as follows:

$$R_{chip} = \frac{N_C}{\Phi_0} A_{R0} + \frac{N_C/\Phi_0 - 1}{\Phi_{\geqslant 1} - 1} A_{R\geqslant 1} + \alpha \tag{5.6}$$

Substituting R_{chip} from (5.6) and A_N from (5.9) into (5.2) then gives:

$$\begin{aligned} A_{chip} =& N_C(N_N(A_\oplus + N_{S_L} A_{S_L}(N_C, \Phi_0, \Phi_{\geqslant 1}) + N_{S_F} A_{S_F} + R_N) + R_C) + \\ & \frac{N_C}{\Phi_0} A_{R0} + \frac{N_C/\Phi_0 - 1}{\Phi_{\geqslant 1} - 1} A_{R\geqslant 1} + \alpha \ , \end{aligned} \tag{5.7}$$

where Φ_N is the fan out of a router level.
Assuming the area of the Pulse Extender, the Differential Pair Integrators (DPIs), an appropriate portion of the MUX block, and the Learning and Stop Learning circuits can all be counted within $A_\oplus$, and that anything else (e.g. wiring) that would have been counted within R_N is negligible, we can set the latter to 0. R_C can cover the AER Row Decoder, Encoder/Decoder, SRAM, Append, Routing Logic, and +1 blocks and will depend on the numbers of cores, neurons per core, and any synapses that rely on the routers S_{HR}, per neuron:

$$R_C = r_C(N_C, N_N, N_{S_{\text{HR}}})$$

Thus the final area cost function (ignoring α) is:

$$\begin{aligned} A_{chip} =& N_C(N_N(A_\oplus + N_{S_L} a_{S_L}(N_C) + N_{S_F} A_{S_F}) + \\ & r_C(N_C, N_N, N_{S_{\text{HR}}})) + N_C A_{R1} + (N_C - 1) A_{R\geqslant 2} \end{aligned} \tag{5.8}$$

For each new hardware, what is necessary to define to get area values from this cost function are:

$A_\oplus$	The area of the pulse extender, DPIs, MUX, soma, learning, and stop learning costs. per neuron.
N_{S_L}	The number of *learning* synapses per neuron.
a_{S_L}	The area of a *learning* synapse.
A_{S_F}	The number of *fixed* synapses per neuron.
a_{S_F}	The area of a *fixed* synapse.
$A_{S_{\text{HR}}}$	The number of *router-dependent* synapses per neuron.
$r_C(N_C, N_N, N_{S_R})$	The area of the AER Row Decoder, Encoder/Decoder, SRAM, Append, Routing Logic and +1 blocks per core per neuron as a function of the number of cores, number of neurons per core, and number of synapses that are router dependent.
A_{R0}	The area of an R0 router.
$A_{R\geqslant 1}$	The area of an R1 or higher router.
Φ_0	The fan out of R0 router.
$\Phi_{\geqslant 1}$	The fan out of R1 or higher router.

Once these are known, one can plot A_{chip} as a function of the number of cores N_C while holding the number of neurons per core N_N constant or as a function of the number of neurons per core while holding the number of cores constant.
This is a general formulation and can be considered for any architecture. Now, we can add assumptions regarding the routing architecture, how synapse types are implemented, and so on.

5.1.2. Specifying parameters in the area cost function

We can now incorporate specific restrictions regarding the architecture of the hardware we are designing in the cost function. All the area values specified here are normalized by the size of a CAM cell. This normalization allows us to compare the size of the current design with already produced chips. The size of the CAM cell used is $3.85\mu * 4.945\mu$.

Synapse areas

With the proposed hierarchical routing, the area occupied by some synapse types depends on the fan-outs of the routers and the depth, h, of the router tree. Given that h is itself a function of N_C, Φ_0 and $\Phi_{\geqslant 1}$, the area $A_{S_{\text{HR}}}$ of a hierarchical routing synapse is a function of these three variables:

$$A_{S_{\text{HR}}} = a_{S_{\text{HR}}}(N_C, \Phi_0, \Phi_{\geqslant 1})$$

Adding cores will affect the area occupied by these synapses.
In this hierarchical design, besides the learning and fixed synapses, the area of a synapse also depends on the distance among neurons (or to which router level they need to reach). This brings us to other two categories of synapses: *External synapses* (that represent long-range connections), which participate in the hierarchical routing scheme, that will be described with the subscript S_E; and those called *Recurrent synapses* which do not participate in the hierarchical routing described with the subscript S_R. Figure 5.1 shows the external and local synapses considered in this design. With these two synapse types, equation (5.3) for the area of a neuron becomes

$$A_N = A_{\oplus} + N_{S_E} a_{S_E}(N_C, \Phi_0, \Phi_{\geqslant 1}) + N_{S_R} A_{S_R} + R_N \tag{5.9}$$

Local synapses In this architecture, we use simple switches to control hardwired connections in a local core. N_{S_R} should be equal to N_N. Every neuron will have N_N rows for local core connection. Some delay on the handshake circuits of the neuron output can be added, so every row only has two bits latch to choose different DPI and 1-bit latch to control on/off. This way, $N_{S_R} A_{S_R} = N_N * 3 * A_{latch}$, where A_{latch} is 1.508 CAM cells. What leads to

$$N_{S_R} A_{S_R} = 4.5247 * N_N \tag{5.10}$$

External synapses The external connections, as explained before, participate in the hierarchical routing scheme; thus, $N_{S_E} A_{S_E}$ are dependent on N_N and on N_C. The external synapses are subdivided into two types: the ones that contain complete addresses bits, i.e., are capable of creating specific connections between neurons, that will be defined by N_{full} and A_{full}; and the ones that have a constrained number of address bits, i.e., since they can't reach all the routers, they use fewer bits to differentiate the address space. These will be defined by $N_{constrained}$ and $A_{constrained}$.
This way, $N_{S_E} A_{S_E} = N_{full} A_{full} + N_{constrained} A_{constrained}$.
$N_{S_E} = \log_4 N_C + N_{full}$, where N_{full} is the number of full address rows.
For each full address synapse

$$A_{full} = [\log_2(N_C + N_N) A_{\text{CAM}} + A_{PE} + 2 * A_{latch}] \tag{5.11}$$

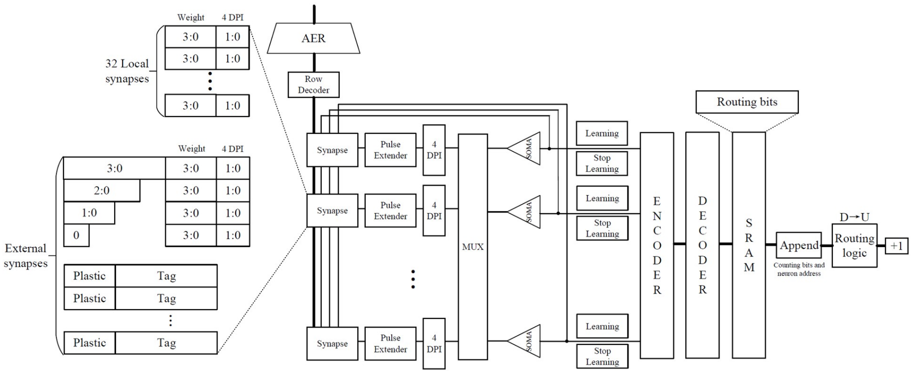


Figure 5.1.: Basic core architecture proposed in the hierarchical scheme.

Any other synapse will depend on the router level, then $A_{constrained} = [\log_2 N_N + (\log_2 N_N - 1) + (\log_2 N_N - 2) + \cdots + 1]A_{\mathrm{CAM}} + \log_4 N_C * [A_{PE} + 2 + A_{latch}]$.
This formulation raises a conflict: if we assume the CAM row for the furthest core is $\log_2 N_N$ bits, then we have already added some constraints on N_C and N_N. $\log_2 N_N$ should be equal to $\log_4 N_C$. Assuming the CAM row for the furtherst core is $\log_4 N_C$ bits, then

$$A_{constrained} = [(\log_4 N_C + 1)\frac{\log_4 N_C}{2}]A_{\mathrm{CAM}} + \log_4 N_C[A_{PE} + 2 * A_{latch}] \tag{5.12}$$

$A_{PE} = 14.8568$ CAM cells. This way, the total cost of external synapses, with all the substitutions, is given by:

$$\begin{aligned} N_{S_E}A_{S_E} &= N_{full}A_{full} + N_{constrained}A_{constrained} \\ &= N_{full} * [\log_2(N_C + N_N) + 17.8728] + \\ N_{constrained} &* \left\{[(\log_4 N_C + 1)\frac{\log_4 N_C}{2}] + \log_4 N_C * 17.8728\right\} \end{aligned} \tag{5.13}$$

Other synapses types Every synapse will have two bits to choose one of the four different synaptic types available (AMPA, NMDA, GABA$_\mathrm{A}$, GABA$_\mathrm{B}$); This way, equation 5.4 will become:

$$A_{chip} = N_C(N_N(A_{\oplus} + N_{S_t}A_{S_t} + R_N + A_{SRAM}) + R_C) + R_{chip} \tag{5.14}$$

where A_{SRAM} is the area of a SRAM word. In this architecture, the area of the SRAM is not added to R_C because every neuron has a SRAM word for routing.

$$A_{SRAM} = A_{SRAM_{cell}} * (4\log_4 N_C + \log_2 N_N) \tag{5.15}$$

Specifically, the SRAM cell area $A_{SRAM_{cell}}$ is 0.7848 CAM cells, thus we can reduce the equation to:

$$A_{SRAM} = 0.7848 * (4\log_4 N_C + \log_2 N_N) \tag{5.16}$$

Soma

In our architecture, the soma has a fixed size: $A_{\oplus} = 336.251$ CAM cells.

Routing levels

The hierarchical routing architecture used here is being designed by Zhe Su and has been explained in Chapter 3. As shown in Figure 5.2, this architecture contains different levels of the router: $R0$ connecting neurons inside the same core, $R1$ connecting a fixed number of cores, and $R2$ connecting $R1$ routers, and so on.
For $R \geqslant 1$, the fan-out is a power of two. This means that if we want to add more cores per R1 router, if the router has $\Phi_1 = 4$, then we will need to add four more cores per R1 router. Similarly, for the next level of routers, the R2 routers, we can not add it with a single $R1$ router. And so on all the way up the tree. Assuming that all routers above R0 have the same fan-out, $\Phi_{\geqslant 1}$, the height of the router tree is given by

$$h = 1 + \log_{\Phi_{\geqslant 1}} \frac{N_C}{\Phi_0}$$

The functions r_{chip}, r_C and r_N, which include the areas occupied by the routing, will have to be written in terms of h and/or Φ_0 and $\Phi_{\geqslant 1}$. And the area of the synapses also depends on h, Φ_0, and $\Phi_{\geqslant 1}$ in this hierarchical design.

Routing overheads

Routing cost per neuron This architecture's routing cost per neuron does not depend on other variables since it is only a handshake module. This way, R_N is a constant.
$R_N = 71.229$ CAM cells.

Routing cost per core R_C is still being designed and is mainly decided by the cost of arbiters $A_{arbtree}$. The arbitration mechanism was optimized to a hierarchical arbitration with fewer arbiters and faster speed. The area of the basic element in the arbiter tree now is $A_{arbtree} = n*(N_N^{\frac{1}{n}}-1)$, where n is the hierarchical depth.
Usually, the hierarchy depth is $n = \log_4 N_N$. This way, the arbiter number is $3*\log_4 N_N$. This leads to $R_C = 3*\log_4 N_N * A_{arbiter} + \beta$, where β is a constant representing other asynchronous pipeline circuits and encoder. Although they also change as N_N changes, they are not a critical part of the area, so they can easily be ignored. The arbiter area $A_{arbiter} = 8.1896$ CAM cells. Thus

$$R_C = 24.5688*\log_4 N_N \tag{5.17}$$

Routing cost per chip The cost of routing per chip R_{chip} will increase as N_N and N_C increase. N_N affects the bus data width. N_C will also affect the bus data width and the router number. In this architecture, by choosing four leaves nodes (fan-out) per router $\Phi_{\geqslant 1} = 4$, it is possible to optimize the data packet size, which affects source memory size, dynamic routing power and NoC architecture area cost. And because of the mask routing used at every level, the number of leaves nodes per router does not have to be a power of two. But considering source memory size, dynamic routing power, hierarchical depth, hierarchical design flow, and NoC architecture area cost, four leaves nodes per router seems like the best choice.
With this, we can simplify the calculation of the area cost, and we can ignore the increase of router area by changes in the NoC architecture.
This way,

$$\begin{aligned} R_{chip} =& A_R*(N_R+1)+\alpha \\ =& A_R*\left(\frac{\frac{N_C}{4}\left(1-\frac{1}{4}^{\log_4 N_C-1}\right)}{1-\frac{1}{4}}\right)+\alpha \\ =& A_R*\frac{N_C-1}{3}+\alpha\ , \end{aligned} \tag{5.18}$$

where α is a constant representing other asynchronous pipeline circuits on the long wires, biasgen, and other interface circuits.
The router area is $A_R = 22,768.10$ CAM cells. Thus,

$$R_{chip} = 7589.366*(N_C-1)+\alpha \tag{5.19}$$

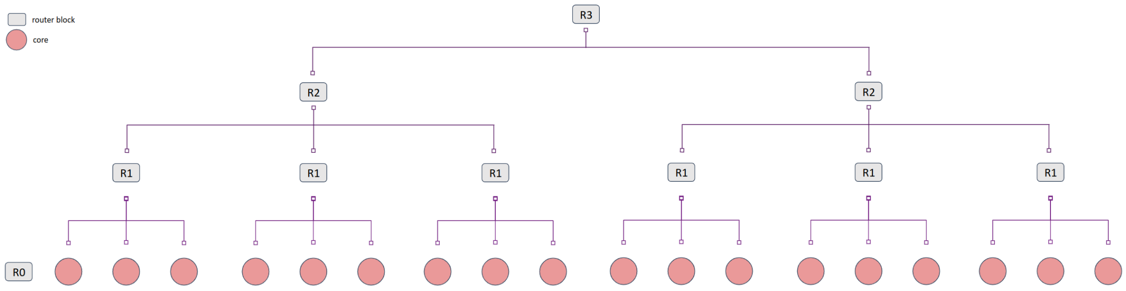


Figure 5.2.: Schematic of the hierarchical routing scheme designed by Zhe Su.

Total costs

With all the variables defined, we can put all of them together. The final cost function for this architecture depends on four variables: N_N, N_C, N_{full} and $N_{constrained}$; and two constants: α and β.
This way, the final cost function can be written as:

$$\begin{aligned}
A_{chip} &= N_C(N_N(A_{\oplus} + N_{S_t}A_{S_t} + R_N + A_{\mathrm{SRAM}}) + R_C) + R_{chip} \\
&= N_C\Big\{N_N\Big\{336.251 + 4.5247 * N_N + \\
&N_{full} * [\log_2(N_C + N_N) + 17.8728] + \\
&N_{constrained} * \Big\{\left[(\log_4 N_C + 1)\frac{\log 4N_C}{2}\right] + \log_4 N_C * 17.8728\Big\} + \\
&71.229 + 0.7848 * (4\log_4 N_C + \log_2 N_N)\Big\} + \\
&24.5688 * \log_4 N_N + \beta\Big\} + 7589.366 * (N_C - 1) + \alpha
\end{aligned} \tag{5.20}$$

Additonal Constraints

We could additionally consider other types of constraints, for example, the "powers of two constraints". This constraint would enforce the size of cores, and the number of cores follows a "power of two" structure (a core could have 2, 4, or 8 neurons but not 3 or 5, for instance). For this, we have to replace N_C and N_N with 2^c and 2^n respectively, as in (5.21), and vary c or n while keeping the other constant instead.
Then we can define $c = \lceil \log_2 N_C \rceil$, $n = \lceil \log_2 N_N \rceil$ and $s_t = \lceil \log_2 N_{S_t} \rceil$ and rewrite equation (5.4) as

$$A_{chip} = 2^c(2^n(A_{\oplus} + \sum_{t=1}^{T} 2^{s_t} A_{S_t} + R_N) + R_C) + R_{chip} \tag{5.21}$$

Number of bits per neuron

The number of bits per neuron depends on the number of router levels and the number of rows available for each router level.
The connections inside a core are hardwired, and there is no need for extra memory bits. Connections from an $R1$ router are defined for a single bit: a neuron receives all the spikes from the same half of all the cores reached through an $R1$ router. Connections from an $R2$ router need two bits to be described: we need to specify the fourth of each core that can be reached. For $R3$ routers, each neuron needs three bits, and so on.
This way, the number of bits per neuron depends on the number of router levels[3]

$$\text{Bits per neuron} = \sum_{i=1}^{R} r * i \tag{5.22}$$

r is the number of rows per router level, and R is the maximum router level. Each neuron has one row per router level in our proposed architecture, as explained in Chapter 3, Figure 3.2.
Thus, the number of bits per neuron is defined by:

$$\text{Bits per neuron} = \sum_{i=1}^{R} i \tag{5.23}$$

[3] The number of router levels is directly related to the number of neurons per core. For cores of 16 neurons, for instance, there is no need to have a $R5$ router since no connection could be formed using it.

5.1.3. DYNAP area cost function

DYNAP has four cores, with 256 neurons per core and 64 synapses per neuron.
We normalize the area of DYNAP-SE to the area of a synapse (SRAM area). Since it is a processor already available, the values for the area are known:

- synapse area: 1
- neuron area: 78
- core area: 24,164
- 4-core (chip) area: 118152

The area in the DYNAP-SE grows not linearly because of the interface circuits, router circuits, and some other redundant areas.

DYNAP bits per neuron

DYNAP-SE paper (Moradi et al., 2018) describe the number of bits per neuron given the following equation:

$$\text{Bits per neuron} = 2 * \sqrt{F * \log_2 C * \log_2 N} \tag{5.24}$$

where,

- F is the fan out per neuron (number of outgoing connections)
- C is the number of neurons per core
- N is the total number of neurons

However, it assumes an ideal M (the fan in per neuron) that is not fit for our case. In this case, a more accurate equation is given by the point where $K == C$, and K is the number of tags available per core in DYNAP-SE:

$$\text{Bits per neuron} = \frac{F}{M} * \log_2 N + M * \log_2 N_C \tag{5.25}$$

where,

- F is the max fan out per neuron (number of outgoing connections)
- M is the max fan in per neuron (number of incoming connections)
- N is the total number of neurons
- N_C is the number of neurons per core

5.1.4. TrueNorth area cost function

The physical layout of a TrueNorth (Merolla et al., 2014) core in $28nm$ CMOS has 240 μm×390 μm. The neuron area is 2900 μm^2, with an additional area of 3 μm^2 to store the neuron state.

TrueNorth bits per neuron

In TrueNorth, each neuron can target any axon on any core up to 255 away, i.e., up to 4 chips, in both x and y dimensions. This is the equivalent of 17 billion synapses; therefore, it uses 26 address bits per neuron.

Each neuron in TrueNorth has a maximum fan out of 256 neurons. When the fan out of a neuron increases over this limit, we need to make use of a relay neuron, i.e., we need to use a neuron as a proxy for the remaining connections.

Although the number of bits per neuron does not increase with an increase in the fan out, it does impact the total bit memory needed to map a network.

An additional relay neuron can lead to the use of a new core, thus, 256 × 26 extra bits. Note that the increase in the number of relay neurons can amortize this additional memory (by adding them to the same core) until a new core needs to be used again.

To compare the total amount of memory when mapping the same network to each one of the architectures, for TrueNorth, we calculate the number of extra cores needed by the relay neurons:

$$\text{extra cores for relay neurons} = \lceil F \mod 256 \rceil / 256 \tag{5.26}$$

Where, F is the fan out per neuron.

5.1.5. Area comparison

The area for this work is calculated by using the cost function presented in Equation 5.20. There are a few parameters that we need to define to provide a specific value for area: the number of neurons per core and the number of cores. In order to offer a fair comparison, the number of neurons per core is fixed in 256 as in TrueNorth and DYNAP. For this architecture and TrueNorth, we assume all the cores can be placed on a single chip. DYNAP has an architecture with four cores per chip, so we consider the total area size as the sum of the area up to three cores, and when four cores are needed, we consider the area of a chip.

Our analysis shows that considering current technology, a chip to fit a network with a million neurons and a fan-in/fan-out of roughly a thousand per neuron would require an area of $\approx 0.9^{10e7}\,\text{mm}^2$ if implemented with our scheme, about 5x more on the DYNAP ($\approx 4.49^{10e7}\,\text{mm}^2$) and almost 3x more on the TrueNorth ($\approx 2.68^{10e7}\,\text{mm}^2$), as shown in Figure 5.3.

5.1.6. Bits per neuron comparison

In our design, the number of bits required to fit a network does not increase with the neuron fan-in/fan-out. We can increase the fan-in by adding more cores, so the increase in memory is linear in the number of cores. On the other hand, the TrueNorth architecture has a fixed fan-in per neuron, and with an increase in fan-in/fan-out, we need to recruit relay neurons from other cores. This is costly for large networks in which neurons have a large fan-in. Indeed, any architecture with a fixed fan-in per neuron will not scale well due to the requirement to resort to relay neurons and will consequently increase memory requirements (Rao et al., 2022). Also, the fan-in is fixed in the DYNAP architecture. But its mixed source/destination addressing scheme mitigates the number of intermediate nodes required. Our analysis shows that a network with a million neurons requires 67000 kb if implemented with our scheme, about 10x more on the DYNAP (734000 kb) and about 28x more on the TrueNorth (1904000 kb), as shown in Figure 5.4.

5.2. Experiments description

After the verification that our heuristic works as expected for the canonical network, we design experiments to validate the placement behavior in non-ideal cases. Two canonical WTA networks were considered for these experiments, based on hardware with 16 neurons per core, and either 7

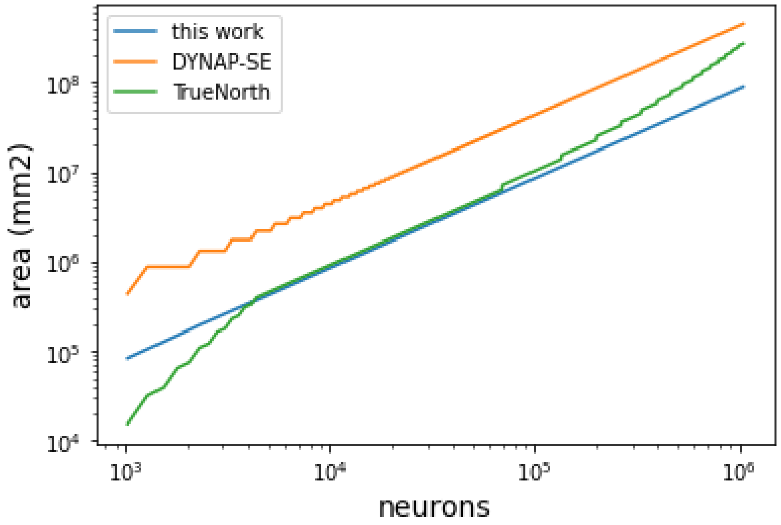


Figure 5.3.: Area scaling comparison between the TrueNorth (Merolla et al., 2014) and DYNAP (Moradi et al., 2018) architectures and this work. For comparison, we considered a core with 256 neurons in all architectures. TrueNorth and DYNAP are existing chips with well-defined chip areas. We extrapolated their area, given the number of cores/chips needed for the placement. The chip area for the three architectures is plotted as a function of the network size.

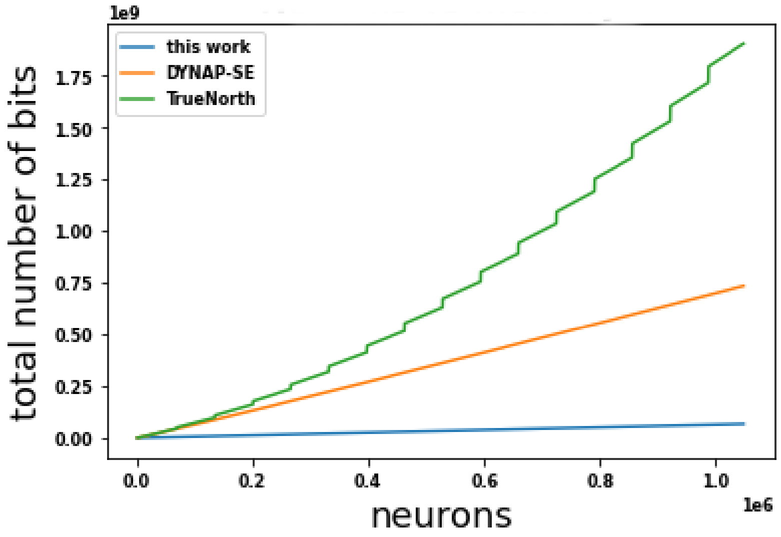


Figure 5.4.: Memory scaling comparison between the TrueNorth (Merolla et al., 2014) and DYNAP (Moradi et al., 2018) architectures and this work. For comparison, we considered a core with 256 neurons in all architectures. The number of bits used by the three architectures is plotted as a function of the network size. For TrueNorth, the number of cores, thus, the number of bits necessary to fit our canonical network, increases approximately quadratically with increasing fan-in/fan-out of the model. For DYNAP, the memory increases linearly with increasing fan-in/fan-out. In our proposed architecture, the increase in fan-in/fan-out demands more cores, but the memory increases slowly.

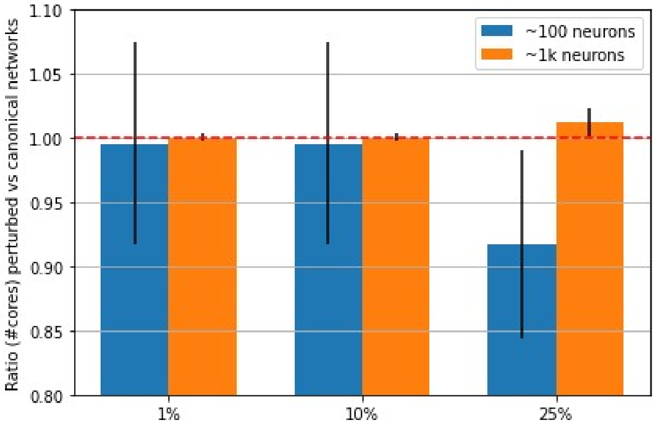


Figure 5.5.: Placement of deviation of our canonical network by removing a fixed percentage of neurons. We created 300 network variations for each canonical network (by randomly removing 1%, 10%, and 25% of neurons), leading to a total of 600 new networks. Each bar shows the average and standard deviation as a ratio of the number of cores needed to place the perturbed networks to the required number given the GT for each set of 100 networks. One means both placements use the same number of cores. The red dashed line marks the canonical network placement. All values are normalized to the size of the canonical network.

or 70 cores, leading to networks with 112 and 1120 neurons in total. The networks were created considering the cores were placed in a 1d-line.

5.2.1. Removing neurons and synapses

First, we tested our algorithm using perturbed networks that deviate from the canonical by removing a percentage of neurons (1%, 10%, and 25%). For each one of the canonical networks and for each percentage of neurons to be removed, we created 100 new networks. Precisely, we created 100 networks by removing 1% of the neurons from the original 112-neuron canonical network, then 100 networks by removing 10% of the neurons from the original 112-neuron canonical network, and again 100 networks by removing 25% of the neurons. The same procedure was applied to the 1120-neuron canonical network. This led to a total of 600 new networks that are variations of the canonical structure.
With this experiment, we evaluate how much deviations from the canonical network affect the placement algorithm. Our algorithm found solutions very close to the GT for minor deviations (1% and 10%). More significant deviations (25%) produce solutions that are close to the GT only in large networks (∼1k neurons). Also, small networks (∼100 neurons) produce more significant variations in the solution space. These results are interesting, as we expected them to be independent of the network size. This is presumably due to the combinatorial search space of possible networks: for each canonical network, we generate 100 network variations for each proportion of neuron removal. Thus, we generate 100 networks from the ∼100-neuron network, which can cover the search space of possible different networks. However, generating 100 networks from a ∼1k-neuron network creates a smaller subset of networks being analyzed.
Figure 5.5 shows the results of this experiment.

Secondly, we also tested the placement algorithm by, again, removing random neurons from the original canonical network. This time, we continuously removed neurons (from 1 to all the nodes in the network). This means we generated 100 new networks from each one of the original canonical networks by continuously increasing the number of removed neurons. This experiment helps us see the placement fluctuation with small step increments. Furthermore, we see that the variation in cores usage differs for different network sizes. For the perturbations created based on the 112-neurons canonical network, we see a maximum variation of 30% in the number of cores needed (compared to the GT). For the larger networks (originated from the 1120-neurons canonical network), the variation in the number of cores to place them is, at maximally, 5% when compared to the GT. The results of the placement experiments are shown in Figure 5.6.

For all perturbed networks and network sizes considered, the performance of the placement algorithm is close to the GT performance, indicating an optimal use of the limited resources on the neuromorphic hardware. And interestingly, the performance gets closer to the GT as the network gets larger and with more considerable variations. Why this happens is not intuitive; however, it is a piece of essential information when we talk about scaling up networks and having them placed automatically on neuromorphic hardware.

5.2.2. Changing network structure

In both previous experiments, we started from networks fitting the hardware and removing neurons (and their synapses) from them. The networks would get smaller, and any populations left would always fit the cores.

We decided to analyze how much the placement would deteriorate if we changed the intrinsic nature of the networks. Again, accessing the optimality of general networks is challenging. So we decided to start again with a canonical network, but instead of removing neurons and synapses, we will randomly swap edges. The changes in connectivity between neurons can lead to substantial structural differences in the network.

Starting with a canonical network (with a total of 112 neurons and 16 neurons per core), we swap an increasing number of edges. By swapping edges, we mean adding and removing edges, given how much we want to change the network. For instance, if we want to change the network structure by 10%. Then we will randomly change n edges, where n is the number of edges in the network times the percentage we are swapping. We randomly select $2n$ pairs of neurons that will lead to n edges to be removed and n to be added[4]. The results of this experiment can be seen in Figure 5.7.

In the previous experiment, we guaranteed that the network would fit the hardware structure provided. By swapping edges, we can now form populations with more neurons than fitting a core or with more connections among populations that our hypothetical processor would accept. These differences in network topology force us to use more cores to fit them. It is really hard to assess the optimality of a network placement for a random network.[5] On average, the placement of the deviations of our canonical network needed 18 cores, which is more than double that necessary for our canonical network. However, it is important to notice that the increase in number of cores, although demading more physical area on a chip do not scale up the memory needed to place the network.

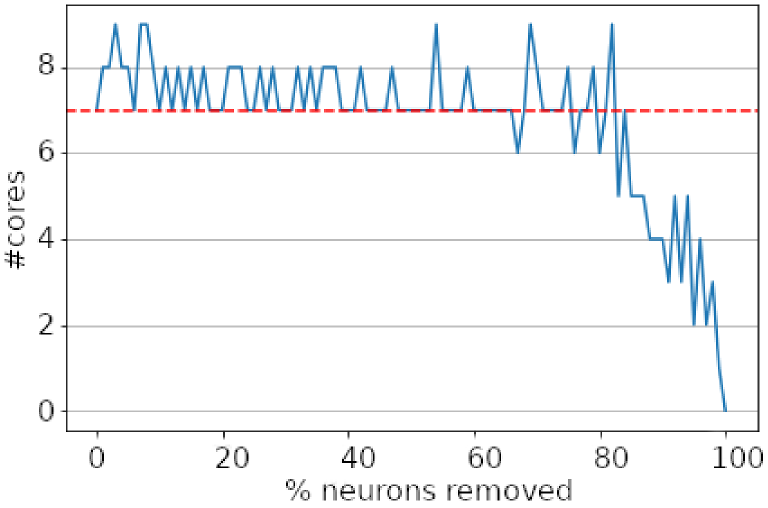
5.3. Placement of a Recurrent Neural Network

RNNs were designed to learn sequential or time-varying patterns. An RNN is a neural network with feedback connections (Medsker and Jain, 2001). Their architectures range from fully interconnected neurons without distinction of input layers to partially connected with distinct input and output layers.

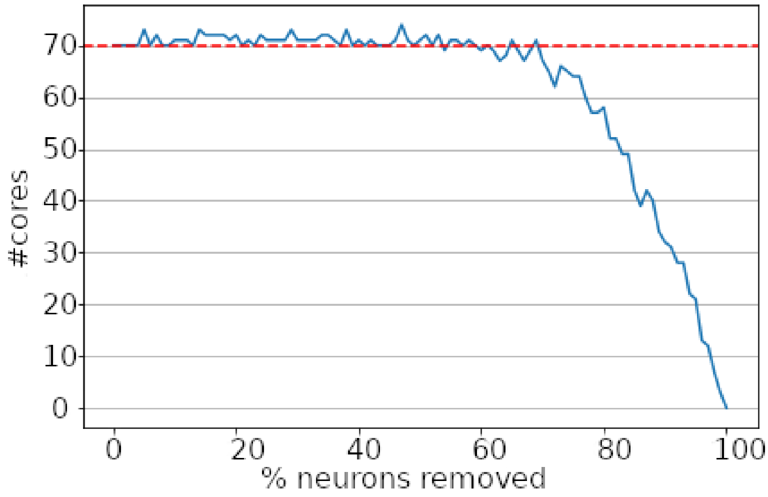
Neural oscillations are a fundamental mechanism that enables the synchronization of neural activity within and across brain regions and promotes the precise temporal coordination of neural processes underlying cognition, memory, perception, and behavior (Neustadter et al., 2016).

[4] Note that we can "add" an existing edge and "remove" a non-existing edge.

[5] We explain in the Chapter 6 that we can calculate the minimum number of clusters and bits necessary for any network following information theory analysis. If a solution exists, we can figure out how far from it we are.



(a)



(b)

Figure 5.6.: Placement of deviations of our canonical network by removing an increasing percentage of neurons. The red dashed line marks the canonical network placement. For each one of the canonical networks, we remove an increasing number of neurons and place the perturbed network. (a) shows the result for the ~100-neuron network, and (b) for the ~1k-neuron network. Note that for the perturbed networks, there is some fluctuation in the number of cores needed; however, it stays close to the ideal case. The Y axis shows the ratio of the number of cores needed to place the perturbed networks to the number required for the canonical network.

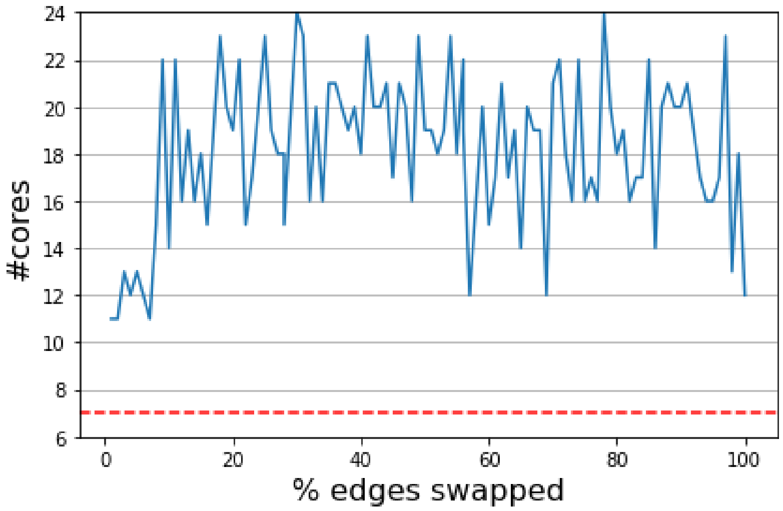


Figure 5.7.: Placement of deviations of our canonical ∼100-neuron network by swapping an increasing number of edges. The red dashed line marks the canonical network placement (i.e., created based on a processor with seven cores). It is not trivial to define what is the optimal placement for a random network. Every new network placed has a different number of random edges added or removed, changing the network's topology. The average number of cores needed to place the deviations of our canonical network is 18, with a standard deviation of 3 cores.

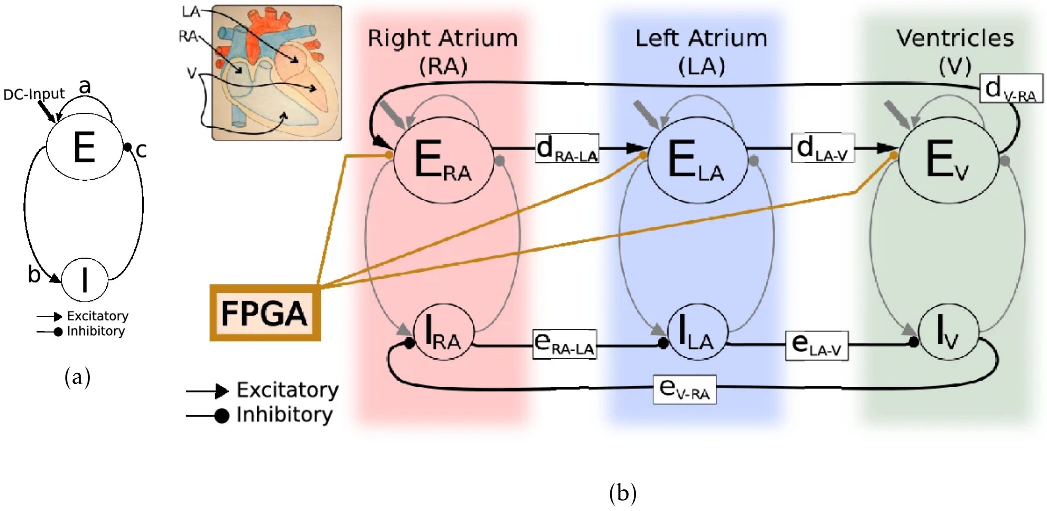
The advent of mixed-signal analog/digital neuromorphic electronic circuits provides new means for implementing neural coupled oscillators on compact, low-power, spiking neural network hardware platforms (Krause et al., 2021). Here, we reproduce this RNN for coupled oscillators to show the validity of our placement. Figure 5.8 shows the structure of the network used.

The network consists of six populations. Three are excitatory, with sixteen neurons all-to-all connected, and three are inhibitory, with four neurons all-to-all connected. The populations with sixteen neurons share 16 connections between populations per neuron, i.e., all the neurons from one population send a connection to every other neuron in the next population. The same happens for the inhibitory populations with four neurons each. Also, excitatory populations send 16 connections to the inhibitory ones, and they send back four connections.

Remember that in our architecture, the number of connections between populations is related to the distance and the number of neurons per core. To share sixteen connections between two populations, the minimum number of neurons per core needs to be 32. With 32 neurons per core, at a distance of 1, we place up to sixteen connections, at a distance of 2, eight connections, at a distance of 3, four connections, and so on. Also, at every distance, a neuron can listen to a subpopulation of all the other cores that can be reached through that router level.

Figure 5.9 shows two approaches to placing this network in our hierarchical hardware structure. In Figure 5.9a, the result of our placement algorithm considers 32 neurons per core. With 32 neurons per core, populations that share 16 connections are placed at distance 1; populations that share 4 connections are placed at distance 3. Between the excitatory and inhibitory populations, the number of connections is non-symmetric. Excitatory populations send 16 connections to the inhibitory ones, which send back four connections. We keep the further distance between them and place the cores at a distance of 3 from each other.

Neurons in cores at a distance of 1 can listen to either the left or the right side of all the cores at that level. The red population listens to the left side of all the cores from $R1$, i.e., the cores with blue and green populations. The second core, containing the blue population, has no neuron active on the left side, so no connection is formed. The blue population is listening to the right side of all the cores from $R1$. Similarly, the third core has the population placed on the right side,



(a)

(b)

Figure 5.8.: RNN used to implement coupled oscillators on neuromorphic hardware. In 5.8a an example of a coupled oscillator consisting of two reciprocally connected inhibitory and excitatory neuron populations. In 5.8b the structure of three neuronal oscillators used by Krause et al. (2021) to model the activation times of the right atrium, the left atrium and the combined ventricles. Figure adapted from Krause et al. (2021).

which doesn't generate any connection. However, the green population should receive connections from the blue population, localized on the left side of the core. If we allow the neurons in the green population to listen to the blue population, they will also receive connections from the red population, which is undesirable. We then mark the connections from blue to green as inconsistent. The inhibitory populations share four connections. This leads to the use of a $R3$ router. At a distance of 3, every neuron can listen to eight different areas, and no connection is marked as inconsistent. This allows all the connections from the inhibitory population to be placed, but only a third of the connections from the excitatory populations. Thus, twelve connections are marked as inconsistent.

With the definitions of inconsistencies, we can loop over them and check if moving a neuron to another position would reduce the number of inconsistencies. But no solution is found. All the inconsistencies are kept.

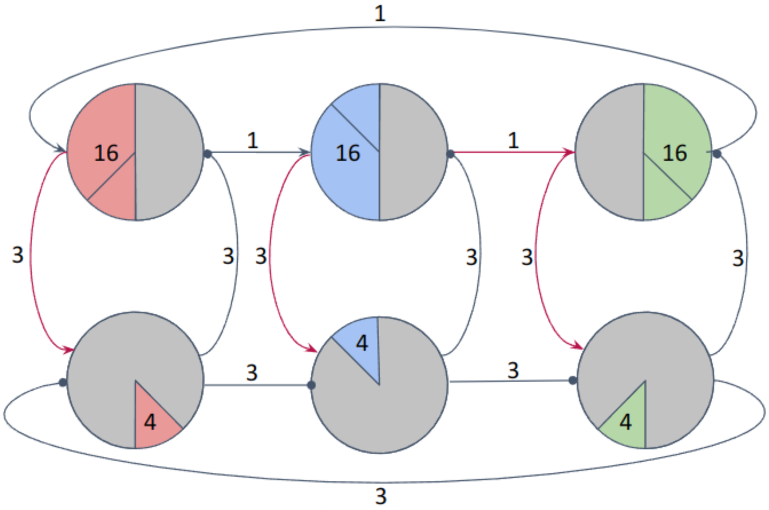
One way to reduce the number of flagged connections (or inconsistencies) during the placement is, for instance, by changing the number of neurons per core. The number of neurons per core is heavily associated with the number of connections that can be formed between two cores.

Figure 5.9b shows the placement result when we increase the number of neurons per core to 64. In this case, the excitatory populations are placed at a distance of 2, given the number of connections among them. At a distance of 2, every neuron can differentiate between four subgroups from all the cores reached through an $R2$ router. This gives us enough differentiation to place all the connections between the excitatory populations. The red population can now listen to the bottom-right area of all cores reached through $R1$, where only the green population is active. The blue population listens to the top-left area of all cores reached through $R1$, thus only the red population. And similarly, the green population listens to the top-bottom area of all cores reached through $R1$, i.e., to the red population. No inconsistencies are generated. Similarly, the inhibitory populations can be placed without inconsistencies through an $R4$ router. Excitatory and inhibitory populations talk to each other through an $R3$ router, which allows every population to listen to eight neurons. All the connections from inhibitory to excitatory can be placed, but only eight from the excitatory to the inhibitory, generating eight inconsistencies.

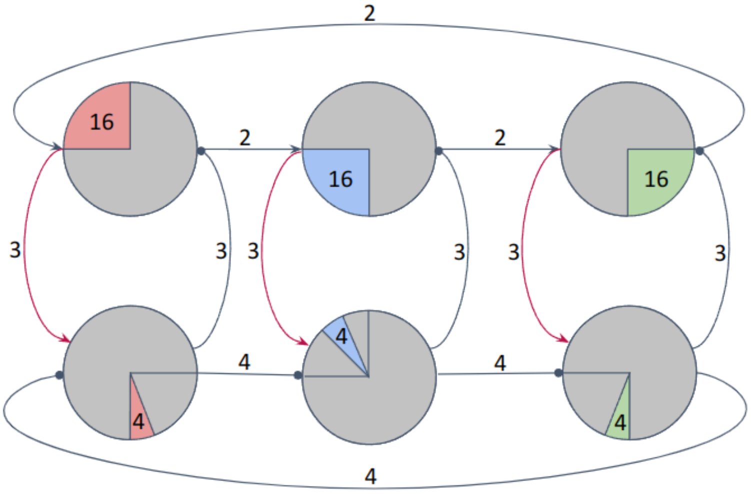
In our current implementation, we flag connections and can use the available full-address rows to add them in order to fit the network. This placement is an excellent example of how vital hardware and software co-design is. Of course, increasing the core size gives us more flexibility in the placement, but we can see a waste of neurons in each core. Other approaches we can take to increase the flexibility of placement in this structure is, for instance, to increase the number of rows per neuron for each router level.

5.4. Summary

Our placement algorithm can place the network perfectly when using a network that matches our hardware design. To evaluate the placement in different conditions, we performed tests where the network topology was modified by either removing neurons or swapping edges. We show that the algorithm can still find a placement for the network in both cases. We also show how the placement works for a RNN application and how the software-hardware co-design approach can help to guide chip design. We can identify compromises and alternative solutions for both hardware and software by evaluating automatic software placement and analyzing the hardware structure. More than that, we also provide a scaling comparison between the memory and the area necessary to place a network using our routing scheme and comparing it with TrueNorth and DYNAP architectures. Since our designed architecture has yet to be available, we don't have accurate measurements of its size. However, we provide an area cost function that can also be adapted to other architectures. It is interesting to notice that our hierarchical routing scheme needs significantly less memory to create network connectivity, but the scaling in the area is not as high. This occurs because of our current technologies for designing the neuron and synapse circuits. Nevertheless, we can still provide an example of how to scale up our chips for networks with high fan-in and fan-out.



(a) Placement on cores with 32 neurons. A core listens to all the other cores that can be reached through the same router. This generates inconsistencies with this placement where the green population should listen to the blue but not to the red one.



(b) By increasing the core size to 64 and having 16 connections between cores at a distance of 2, we can reduce the number of flagged connections.

Figure 5.9.: Results of the placement of the oscillator RNN network. Circles represent the cores. Colored areas mark active neurons that are sending and receiving spikes. The number inside the colored areas show the number of neurons active. Areas with 16 neurons are excitatory populations and with 4 neurons are inhibitory. Values on edges represent the router level the connection has to use. Red edges indicate that some or all connections can not be placed. In 5.9a, the placement considers a core of 32 neurons. In 5.9b, the placement considers a core of 64 neurons. The flexibility in hardware parameter choice can lead to better design choices in the software-hardware co-design framework.

"One never notices what has been done; one can only see what remains to be done."

Marie Curie

6. Discussions

Neuromorphic engineering and computing have emerged as exciting and attractive research areas. Its shift from the conventional von Neumann architecture towards the development of ultra-low power circuits and systems provides excellent advantages in energy efficiency, power consumption, and adaptability to complex tasks that demand interaction with the environment.

Since its creation, neuromorphic engineering has significantly evolved and nowadays not only deals with hardware implementations but also with computing and algorithms.

In recent years, we have seen the creation of a plethora of machine learning algorithms that deal with big data and use massive computer clusters to harvest information. And although this can offer state-of-the-art solutions, they ignore a current challenge: the use of those algorithms in power-critical applications, such as edge computing. Our goal is to have hardware and software that can interact with the environment and sense and process information within constrained resources.

These challenges brought us to the research presented in this Thesis. This Chapter provides a reflection on the research process, the overall impact and implication of its outcomes, some current limitations, and recommendations for future research.

6.1. General remarks

Neuromorphic engineering as a research field integrates a wide range of domains and applications: biology, physics, mathematics, computer science, electronic engineering, robotics, VLSI circuits, and technologies, among others. To mimic brain-like capabilities, the machine-learning community resorts to mining information from big datasets. Mainly, they focus on retrieving and extracting relevant information or patterns from an immense volume of data. The computational resources and power necessary to do so are outstanding and a luxury not found easily in solving real-world problems. Having bio-inspired technology, architectures and algorithms will pave the way for the next breakthrough in understanding the brain.

Both biological and VLSI implementations of neural systems share physical limitations imposed by the machinery where they are implemented, and this machinery has a crucial effect on the computational algorithm used.

The vision that we should be able to create electronic systems that operate like biological brains brought us already to event-sensor systems and offered us new ideas and breakthroughs. However, we still need to overcome the problem raised by the interaction of modules, cores, and chips. Neurons are a particular type of cell, and they communicate with each other through synapses, i.e., they have a direct link. Their nature of communication is both analog and digital. Even more, in the brain, the proportion of white matter (the part of the brain that holds the synapses) tends to grow when the number of interacting modules grows or when the brain size increases (Zhang and Sejnowski, 2000). This also happens for the communication of modules on VLSI systems: the costs of memory and area are higher to create the connections than to processing events. The communication between the neurons in VLSI design is currently a major liming factor for scaling up the systems. The degree of connectivity allowed within chips, and inter-chips highly depends on the structure and approach used (in other words, on the designed routing scheme). Also, given the intrinsic difference in the nature of analog hardware, inter-chip communication needs different approaches than the ones used by traditional fully digital computers. That said, we envision that technologies that allow scaling of the connectivity among cores and chips will dominate, moving in the direction of embracing digital and analog components, as we see in the natural world. This will happen not only by developing new materials and hardware technology

but also by new algorithms and approaches for computation.
By now, the term neuromorphic has been used to describe anything that, even vaguely, relates to brain-like structures. The diverse nature of this neuromorphic research field makes it challenging to analyze and compare different systems. It is a concrete need to have an explicit computational stack, definition of standards, and benchmarks. And that will allow us to build an electronic system that interacts with the environment, reading event information from sensors and processing them in real-time with low energy like the brain does.

6.1.1. The future of neuromorphic systems

One critical insight and implication of this work is to bring awareness to the future directions of neuromorphic computing systems. Neuromorphic computing is bringing a considerable paradigm shift from classical computing to the table.
Standardizing a computing framework will allow analyzing and comparing different neuromorphic systems over real-world applications. With better performance evaluation, we can make better-informed decisions about the materials chosen to implement them, their architecture, and circuit designs. And this advance in hardware implementation can, in turn, bring innovations in the computation paradigm and algorithms we are using today. However, more than developing new technologies or designing new SNN, this new neuromorphic system also needs to account for algorithms, benchmarks, datasets, simulation systems, and a way to have a fair comparison with classical ANN models.
The co-design approach between hardware and software will also impact the design and evolution of neuromorphic systems. We discuss it in more detail in Section 6.2.7.

Learning algorithms

Classical AI algorithms use deep learning and indeed perform impressively (Schmidhuber, 2015). However, the computational power and energy they require are increasing dramatically, and it might become completely unsuitable (a Hardware Revolution, 2018).
Biological brains are the most energy-efficient processor, so it is reasonable that we try to mimic them. A general biological principle of learning is Hebbian Learning: when a presynaptic neuron repeatedly stimulates a postsynaptic neuron, the strength in connectivity between the pre- and postsynaptic neurons increases. This gives rise to Spike-Timing Dependent Plasticity (STDP) learning, where the strength in connectivity increases when the presynaptic neuron spikes consistently before the postsynaptic neuron and decrease otherwise.
However, although biologically plausible, this type of learning does not produce results that can compete with classical ANN learning, using, for instance, backpropagation.
There is a wave to provide new learning algorithms for SNN that can be as efficient as ANN and still biologically plausible, e.g., eProp (Bellec et al., 2020) SpikeProp (Bohte et al., 2000) Deltarule (Mohemmed et al., 2013), ReSuMe (Ponulak and Kasiński, 2010), and others (Dellaferrera et al., 2022; Dellaferrera and Kreiman, 2022).

Benchmarks

The classical AI community has developed extraordinarily in part due to the joint effort of comparing new techniques and algorithms. There are well-defined tasks with a proper response or result called benchmarks. The most common benchmarks available contain extensive labeled datasets.
As a novel computing platform, neuromorphic systems fail to perform and compare with classical AI: the tasks and benchmarks are just unsuitable.
Inspired by biology, neuromorphic systems thrive in resolving real-world problems, notably with low power consumption and latency. We need a new set of benchmarks that also would consider the analog nature of acquiring data from the sensors, how they are stored, and how they are made available to the systems.

We also need a different set of metric evaluations. Accuracy is not the primary performance metric for biological systems, but to solve quickly or to interact with the environment to complete a task.

Simulators

Although simulators are not directly linked with the neuromorphic computing stack, they have a crucial role in helping to explore and design a neuromorphic system.

Hardware simulation is already a standard tool for neuromorphic engineering. Mostly to explore real-world scenarios of data acquisition (or sensors) or the neuromorphic hardware itself, which leads to new algorithms and approaches.

Although simulators do not *perfectly* replicate analog neuromorphic hardware and sensors (it is extremely difficult to take into consideration different sources of noise or data quantization errors, for instance), they provide an easier way to use them without the need to configure or calibrate them, or when the sensors and hardware are not physically available.

6.2. Specific considerations and outlook

Here we will list some additional considerations that are still important for a complete understanding of the scalability of neuromorphic systems. First, we need to point out some simplifications we considered during the development of this work, and then we will briefly describe some possible extensions that can be done moving forward.

6.2.1. Other network types

While small-world networks are our inspiration, we know that the neuromorphic community uses many other network types. Networks that do not follow our imposed structure or deviate too much from it can still be placed but without guarantee of optimally.

While not working on those types, we understand that offering an infrastructure to other connectivity and network architectures is crucial.

In this work, we want to offer inspiration about how to deal with other types of networks. With the available technology, designing general-purpose hardware that can fit the most diverse range of architecture and with a large number of neurons and synapses is not yet reasonable. This work can be extended to deal with other types of connectivities and to help the design of additional hardware.

The network types we envision as the next ones to be used are DNN-types, e.g., Multilayer Perceptron (MLP), CNN, Feed-forward Neural Network (FNN). Those networks are trained using standard gradient descent methods, and their learning process does not take into consideration essential characteristics of natural environments[1]. However, they show significant success in diverse ML tasks, with fast speed and intense computing capacity. Moreover, they can be transformed into SNN that also achieve great accuracy in that tasks (Rueckauer et al., 2017; Sengupta et al., 2019; Stanojevic et al., 2022).

6.2.2. Neuron types

In this work, we abstracted the concept of neuron types. During the development of SNN, some neuron models are commonly used: Hodgkin & Huxley (H&H), Integrate-and-Fire (I&F), Leak Integrate-and-Fire (LI&F), Izhikevich, and others. We are aware that different neuron models can lead to different dynamics. However, for our goal, i.e., placement and routing, a neuron's method to process its event is not a determining factor.

[1] Learning in biological systems does not require a massive set of labeled data, and it is not a separated process of the regular operation of the system

6.2.3. Synapse types

In this work, we also abstracted the synaptic type, i.e., the connection type between two neurons. Most commonly, there are two types of synapses: excitatory and inhibitory. An excitatory connection is where the input spike from a pre-neuron increases the post-synaptic neuron potential. And analogously, an inhibitory connection decreases the post-synaptic neuron potential. Some neuromorphic architectures offer more than just two types of synaptic behavior. For instance, in DYNAP-SE, the excitatory and inhibitory synapses can be subdivided into two further types: fast or slow.

Synaptic types are commonly used during SNN development. This information should also be considered for optimal placement onto neuromorphic hardware and even for designing new routing architectures. The type of synapses allowed for each neuron (and their availability) can strongly impact a hardware's ability to support a desired network.

6.2.4. Weights

The amount of change that will be applied to the post-synaptic neuron is given by the synapse strength, i.e., the weight of the connection.

In a formal ANN, the weight is a scalar value predetermined through an offline training process and remains constant during the inference phase. However, SNNs take a more bio-plausible approach. In SNNs, the synapses experience the phenomenon of plasticity, where the synapse strength is adjustable and found to depend on the relative timing between pre- and post-synaptic spikes of a neuron. This work does not consider the online learning process in our placement and routing structure. Another limitation of our work is our assumption that all synapses arriving at a neuron will have the same connection strength.

6.2.5. Optimality

The optimality of placement and routing for networks into neuromorphic processors is supposed to guarantee that placement and routing can realize the desired network within the given constraints. We should consider all possibilities of placement and routing to guarantee the best possible design. Naively, we could consider generating and evaluating all possible designs or, more specifically, for a given set of networks and hardware constraints to define all possible placements and routings. Then, we could discard the ones that do not fulfill the constraints and pick the one that fits the best, in our case, with the smallest area or memory requirements. Moreover, by following this approach, we could guarantee that the placement and routing for a given set of networks and hardware constraints would either find a solution (satisfying the conditions) or prove the non-existence of a solution (since we would have considered all possibilities). However, an explicit consideration of all possible solutions is infeasible: the absolute number of possibilities would be too massive.

Instead, we use heuristics to find a feasible solution. A feasible solution is a solution that satisfies the constraints. An optimal solution is a feasible solution that results in an optimal value (in our case, minimum memory consumption).

Our approach to designing a new placement algorithm and routing architecture is based on the idea that a connection's cost should consider the distance between the connected neurons, i.e., by using more or fewer bits to define connections depending on how often they are created. This is the same idea behind data compression used, for instance, in jpeg (Wallace, 1992) or morse code (Burns, 2004). These approaches are examples of Shannon's source coding theorem, in which statistical knowledge about the source information can reduce the required capacity of a channel, or, in other words, redundant data can be eliminated from the transmitted information reducing the usage of resources (Shannon, 2001).

A clear direction for future research is to analyze the optimality of our placement and routing. Given that we know the statistics on connectivity (for the network's topology we are focusing on), it is a direct conclusion to compare the number of bits we use with the number of bits Shannon's source coding theorem would give as the best possible solution. Although we can't guarantee to find an optimal solution with our heuristic approach, we could determine how far we are from an optimal solution, if it exists.

6.2.6. Clustering techniques

We use a known algorithm to find our neuron clusters, as explained in Chapter 4. During this Thesis's development, other approaches to cluster neurons were also considered, for instance, cluster neurons by the strength of their connections or even by using k-means, where we could form clusters given the distance between nodes. It would be interesting to analyze how different cluster techniques affect our placement and routing.

Given that our selected architecture is highly structured, another approach we should consider evaluating is based on random walks in the network. The idea of a random walk is that naturally, long paths would be formed inside clusters, and short paths would be given by changing from one cluster to another. This formulation, known as the Map Equation (Edler et al., 2020), is based on information theory and Shannon's source coding theorem.

6.2.7. Hardware parameters

The development of the placement algorithm allows us to play with different hardware parameters: core size, router levels, number of cores per router, etc. Only some of the parameter configurations used in software can lead to a viable implementation of physical hardware. For instance, we saw in the placement of the RNN how increasing the number of neurons per core can reduce the number of flagged connections. However, it does create a waste of neurons that are not being used.

Our digital computers' hardware progress happened because of sustained efforts to meet computation demands as software systems became increasingly integrated into our lives. It is crucial that the process of designing the hardware take into consideration inputs provided by the software framework. Software implementation does not need to follow hard constraints imposed by the hardware. For instance, they can consider an unlimited number of cores per router level or an increasing number of neurons per core. While the physical hardware implementation can not account for such freedom, it can, for instance, learn from the software what parameters have the highest variability and find conditions to move toward them. We can provide neuromorphic processors that accommodate the networks the community needs by using the software system demands to guide new hardware implementation.

"You can't use an old map to explore a new world."

Albert Einstein

7. Conclusion

The development of AI has seen significant progress in recent years, especially with the increasing use of edge computing. However, it pushes for more efficient and specialized hardware to accelerate the performance of edge-computing tasks. Developing domain-specific neuromorphic hardware is one way to advance AI for edge-computing tasks even further. Neuromorphic hardware is inspired by the structure and function of the brain and aims to process information in a more energy-efficient and parallel way than traditional computing. One of the challenges in building large-scale neuromorphic computing systems is optimizing memory resource allocation. In this context, a hardware-software co-design approach can be used to address this challenge. In this Thesis, we presented a co-design approach where brain-like small-world networks inspire the routing scheme and placement algorithm.

Our co-design approach focuses on small-world network topology without limiting its possible applications. Furthermore, focusing on a specific architecture can reduce the necessary memory to place and route networks on a chip. As a result, we can scale up chips with analog design and provide a placement algorithm to find optimal solutions for networks that follow our canonical structure. In this Thesis, we showed that deviations from the canonical design can be placed without diverging too much from the optimal solution. Moreover, the simultaneous design of a placing and routing scheme allows for designing a new multi-core SNN chip that can handle more extensive networks with minimum memory consumption and a smaller area.

In addition to providing a guide to new hardware design, in this Thesis, we also defined the system software, compilers, and a description of a full computational stack for a neuromorphic framework. This work can advance the use of neuromorphic hardware as an emerging computing platform by providing a new computing stack that can be used as a starting point for further development. The next step for the neuromorphic community is a concrete creation of common layers of abstractions in this computing stack. This definition will allow the development of better ways to analyze neuromorphic circuits' performance and compare different hardware on real-world case applications. Even more, these performance analyses can contribute to better-informed decisions about the design of new devices created. Closing the loop in the software-hardware co-design approach is essential to improve our algorithms, software, and hardware in parallel. We can only fully exploit our hardware designs when providing software and algorithms that can make the most out of them.

The best of the best and the worst of the worst
Well, you can never know
The places that I go
I still like you the most
You'll always be my favorite ghost

Florence + The Machine, *Big god*

A. Survey on Networks and Applications

The Neuromorphic Cognitive Systems (NCS) group studies and develops computational models and analog/digital circuits. We surveyed the applications in our group to find patterns and motifs in ANNs. Our goal is to identify characteristics that would help us to develop new neuromorphic hardware that can overcome some of the limitations of the current hardware used in the NCS group. This survey focuses on understanding the networks and applications used in the NCS group. It was created using a google form, and it can be found online[1].

The survey covered applications' type, networks' type, number of neurons, number of incoming and outgoing connections per neuron, etc.

We found that half of our group uses WTA and RNN. WTA are networks that select the maximum from a collection of inputs (Feldman and Ballard, 1982). They are constructed using lateral inhibition among the neurons so that the system is a competitive neural network and can sustain a state. RNN are networks that have a self-sustained temporal activation. This makes RNN dynamical systems. In both types of networks, neurons are connected in clusters and show connections among clusters, as seen in Figure A.1.

Figure A.2 shows the distribution of the network type used in the NCS group.

Besides WTA and RNN, oscillator networks and relational networks also follow a pattern of densely connected clusters with sparse connectivity among clusters.

We also gathered from the survey the type of applications the networks were being used for and what ratio between incoming and outgoing connections per neuron was necessary. The distribution of incoming vs. outgoing connections can be seen in Figure A.3.

[1] https://bit.ly/3dt3wCE

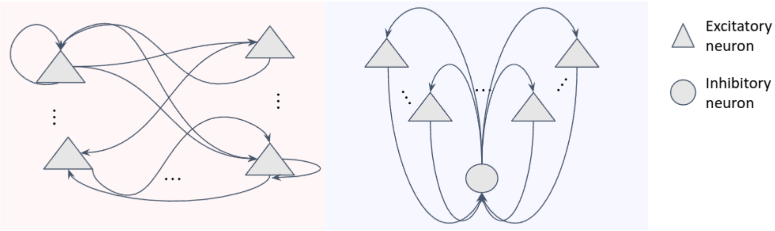


Figure A.1.: Schematics of RNN (left) and WTA (right). In both types of networks, we can identify densely connected groups of neurons and sparse connections between them.

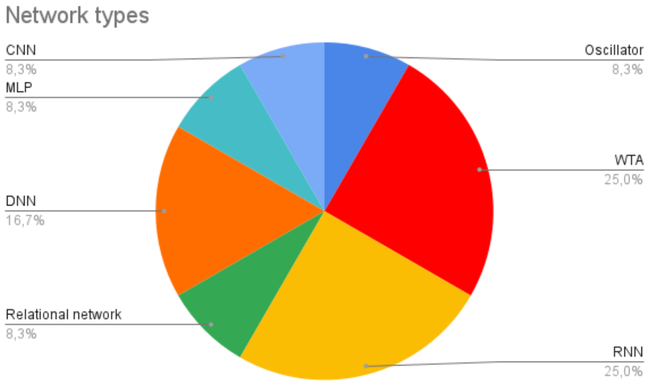


Figure A.2.: Network types identified in the application's survey. WTA and RNN comprise half of the types used (25% each), followed by DNN (around 17%) and CNN, MLP, Relational and Oscillator networks (8.3% each).

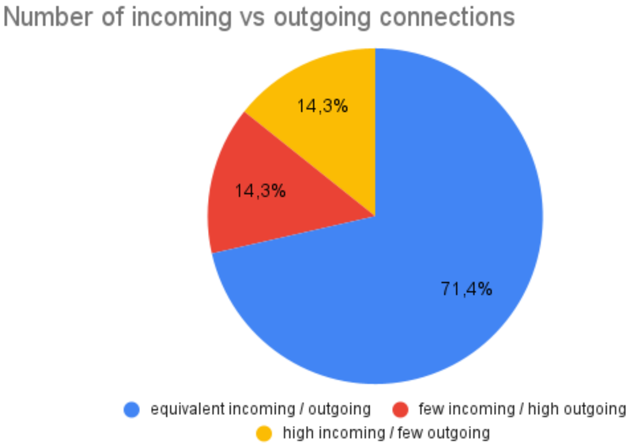


Figure A.3.: Distribution between the ratio of incoming and outgoing connections for all the applications analyzed. For most applications (≈ 75%), having the same number of incoming and outgoing connections is necessary.

The networks in the NCS group can achieve various applications. Table A.1 list the applications collected in the survey and their network types.

WTA and RNN have similar characteristics, and they match the pattern found in biological neural networks, as explained in Section 1.4.3. These network topologies allow a wide range of applications to be modeled; thus, we decided to fix the network's topology to match a WTA structure.

Table A.1.: List of applications, the type of networks, and their relation between the number of incoming and outgoing connections per neuron.

Network type and applications		
Application	Network Type	Ratio of incoming vs outgoing connections
Pacemaker	Oscillator/RNN	equivalent incoming/outgoing
SLAM	WTA	few incoming, high outgoing
Event-based stereovision	WTA	high incoming, low outgoing
Audio and bio-signals processing	RNN	equivalent incoming/outgoing
Emergent behavior/classification	RNN	equivalent incoming/outgoing
Multi-layer Spatio-temporal pattern recognition	DNN	equivalent incoming/outgoing
Motor control, robotics	Relational and WTA	equivalent incoming/outgoing

B. System Software - CortexControl

Introduction

As explained in Chapter 1, neuromorphic hardware provides a platform for efficient real-time simulation of neuronal dynamics and synaptic transmission. However, the configuration and tuning of the neuromorphic chips are still a difficult task.

To alleviate this problem, developing a software toolchain that allows using these chips without knowing all the hardware details is crucial.

DYNAP (Moradi et al., 2018) is a neuromorphic chip developed by SynSense, a spin-off of the INI at the University of Zurich and the ETH Zurich. This chip has been used in the NCS group to develop new computational models. However, it didn't have a software toolchain, and the calibration of parameters on the hardware was non-trivial.

We developed a framework to facilitate the use of the chip, using a modular design approach, envisioning that it could be used and adapted for other neuromorphic chips.

Hardware parameters

The DYNAP-SE processor contains four chips. Each chip is composed of four interconnected cores with 256 neurons. Thus, a DYNAP-SE board contains 4096 neurons.

The neurons are connected through synapses. In the DYNAP-SE, each neuron can be connected to 64 other neurons using four connection types: slow excitatory, fast excitatory, slow inhibitory, and fast inhibitory.

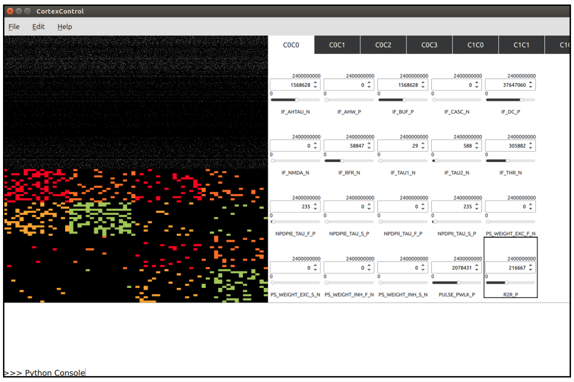
When a neuron spikes, an event is generated and transmitted through digital routers to the synapse circuit. The event contains information about the source (the neuron that produced it) and the target address (the neuron that will receive it). The synapse filters the event, and a proportional current is supplied or absorbed from the connected neuron based on the synapse type.

The connections between neurons are made through the use of a colocalized memory in the neuron, and these are the CAMs and SRAMs. Each neuron contains 64 CAMs and 4 SRAMs cells on DYNAP-SE. All synapse connections have the same weight. Different connection strengths between neurons can be achieved by setting the same connection multiple times.

Software

CTXCTL is a system software aiming at facilitating the control and execution of experiments using neuromorphic hardware platforms that communicate through AER.

Using the GUI of CTXCTL is possible to read and configure the hardware parameters and the neurons' activities. Figure B.1 shows the GUI. The top left part (top half of the black background) shows the raster plot of all neurons (spikes of every neuron over time). Every neuron in the chip is represented as a single row. The bottom left part (bottom half of the black background) shows the instantaneous spiking activity per neuron. For the instantaneous activities, the neurons are ordered as they are physically on the board: 256 neurons per core and four cores. Each core in a chip is represented by a different color. The right side of the image shows the tabs to configure the parameters of each core in each chip ($C0C0$ represents the core 0 from chip 0). For every tab, interface controls such as text fields and sliders can easily configure the different parameters in the hardware. The bottom part of the image shows CTXCTL Python console that allows the users



IF_AHTAU_N 1568628 (0–2400000000)
IF_AHW_P 0 (0–2400000000)
IF_BUF_P 1568628 (0–2400000000)
IF_CASC_N 0 (0–2400000000)
IF_DC_P 37647060 (0–2400000000)
IF_NMDA_N 0 (0–2400000000)
IF_RFR_N 58847 (0–2400000000)
IF_TAU1_N 29 (0–2400000000)
IF_TAU2_N 588 (0–2400000000)
IF_THR_N 305882 (0–2400000000)
NPDPIE_TAU_F_P 235 (0–2400000000)
NPDPIE_TAU_S_P 0 (0–2400000000)
NPDPII_TAU_F_P 0 (0–2400000000)
NPDPII_TAU_S_P 235 (0–2400000000)
PS_WEIGHT_EXC_F_N 0 (0–2400000000)
PS_WEIGHT_EXC_S_N 0 (0–2400000000)
PS_WEIGHT_INH_F_N 0 (0–2400000000)
PS_WEIGHT_INH_S_N 0 (0–2400000000)
PULSE_PWLK_P 2078431 (0–2400000000)
R2R_P 216667 (0–2400000000)

>>> Python Console

Figure B.1.: GUI of CTXCTL. Through the interface, the SNN developer can read the status of each neuron in the hardware and configure the hardware parameters.

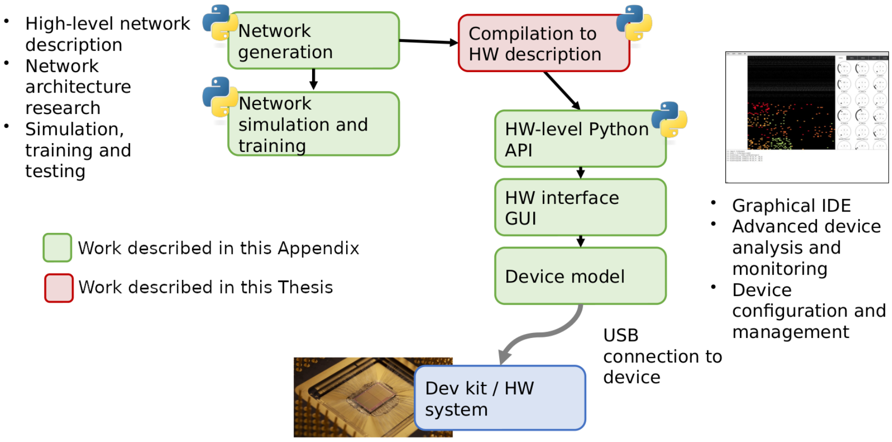

High-level network description
Network architecture research
Simulation, training and testing
Network generation
Network simulation and training
Compilation to HW description
HW-level Python API
HW interface GUI
Device model
USB connection to device
Dev kit / HW system
Graphical IDE
Advanced device analysis and monitoring
Device configuration and management
🟩 Work described in this Appendix
🟥 Work described in this Thesis

Figure B.2.: CTXCTL allows abstracting the hardware by wrapping the libCAER, making the hardware parameters transparent to the user. Through a Python Application Programming Interface (API), SNN developers can easily create their models and apply them to the hardware device that will communicate with the lower-level drivers and firmware. The green blocks were developed as a concurrent project to this Thesis. One implementation of the red block is described in this Thesis, in Chapter 4.

to interact with the chip using python scripts. The Python console is also available in a 'headless' mode, where the GUI is not instantiated.

CTXCTL is developed in C++ and aims to provide three different levels of functionalities. At the basic (lowest) level, all the libCAER[1] functions are available. At this level, to execute a network on hardware, the SNN developer needs to write every piece of information to the hardware, i.e., the neurons' addresses, the definition of a spike's routing scheme, etc. To reduce the effort in executing experiments on hardware, CTXCTL provides an intermediate level where utility functions can be used for neuron instantiation and creating connections between neurons. At this level, the user doesn't need to know the full address of the neuron in the hardware but still needs to choose in what chip and core they will be allocated and connect them using higher-level functions that hide the routing scheme. The third and upper level of CTXCTL allows the description of SNN in terms of neuron population and connectivities without the need to define chips and cores to place the neurons. The block diagram of CTXCTL is shown in Figure B.2.

Figure B.3 shows a demo of CTXCTL live: by configuring a SNN on hardware and stimulating

[1] https://gitlab.com/inivation/libcaer

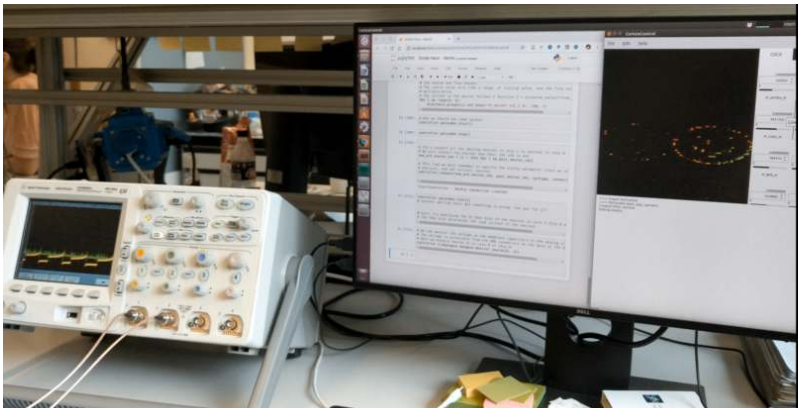


Figure B.3.: CTXCTL being used to configure a SNN, including parameters, input connections, and multi-layer connectivity schemes, implemented using a Jupiter Notebook for Python. The Jupyter Notebook can be used together with CTXCTL as a server option is made available. The oscilloscope on the left shows analog and digital measurements from the neurons on the chip that can be configured using the same setup.

the correct silicon neurons in the chip a "smiley face" is shown on the 2D-surface of the chip. The SNN was configurated so that the input layer projected topographically to a second layer, and then they activate corresponding neurons in the chip. Besides providing a Python Console inside the GUI, CTXCTL API could be used externally, as in this case, through a Jupyter Notebook, by activating a server inside the interface.

CTXCTL is fully made available (Cor, 2020). In-depth documentation of CTXCTL can be found online [2].

Currently, CTXCTL is called Samna, and it is still under development [3].

[2] https://bit.ly/3dnRfzf

[3] https://www.synsense-neuromorphic.com/products/samna/

C. Xinyue Yao's MSc thesis

Is imposing the distance constraint to ANNs reasonable? This was the driving question for the Master project of Xinyue Yao, where Stan Kerstjens and I collaborated to guide the discussions, and Giacomo Indiveri was the formal supervisor. In this project, we analyzed the impact of constraining connections during learning a task in ANN. The distance constraint was implemented as a penalty term in the loss function, so the weight (thus, the connection) between far away neurons would get more substantial penalties during training. With this distance constraint, we could generate networks with major weight distribution towards short-range distances that could still perform their tasks. Then, we analyzed the topology of these distance-constrained sparse networks and found a right-skewed distribution of connections over distance, in which connections decrease almost exponentially from the short-range to the long-range. The distance-modulated connectivity found in this work may be extended as guidance in neuromorphic hardware designs to determine the long-range and short-range connections required in the network and the allocation of memory resources in the chips. Specifically, our findings can reduce the number of long-range connections needed for the network, which may decrease the number of memory cells required per neuron to store both source addresses and tag information. This work has the potential to offer a generic model-based memory optimization method in neuromorphic hardware designs.

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Universität
Zürich UZH

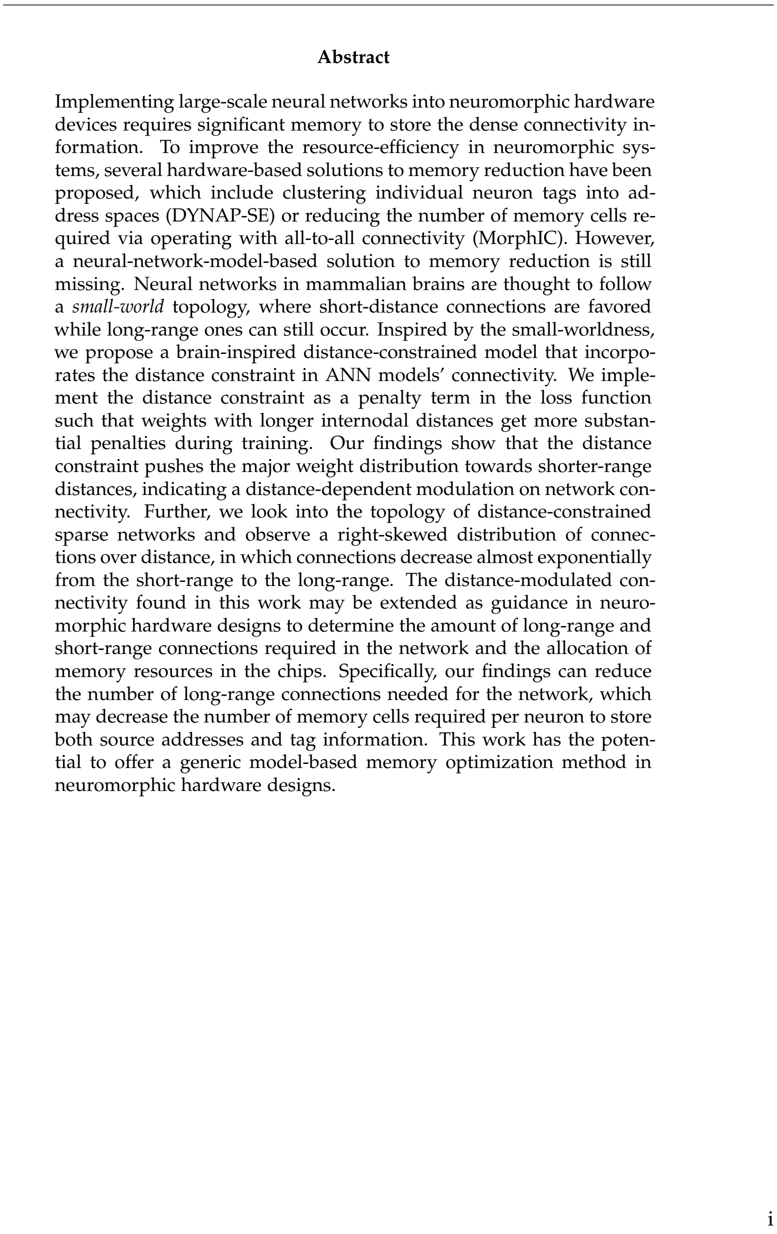
Implementing a Brain-inspired Distance Constraint into Artificial Neural Networks

Master's Thesis

Xinyue Yao

Institute of Neuroinformatics
University of Zurich
ETH Zurich

Supervisors:
Vanessa Leite
Prof. Dr. Giacomo Indiveri



Abstract

Implementing large-scale neural networks into neuromorphic hardware devices requires significant memory to store the dense connectivity information. To improve the resource-efficiency in neuromorphic systems, several hardware-based solutions to memory reduction have been proposed, which include clustering individual neuron tags into address spaces (DYNAP-SE) or reducing the number of memory cells required via operating with all-to-all connectivity (MorphIC). However, a neural-network-model-based solution to memory reduction is still missing. Neural networks in mammalian brains are thought to follow a *small-world* topology, where short-distance connections are favored while long-range ones can still occur. Inspired by the small-worldness, we propose a brain-inspired distance-constrained model that incorporates the distance constraint in ANN models' connectivity. We implement the distance constraint as a penalty term in the loss function such that weights with longer internodal distances get more substantial penalties during training. Our findings show that the distance constraint pushes the major weight distribution towards shorter-range distances, indicating a distance-dependent modulation on network connectivity. Further, we look into the topology of distance-constrained sparse networks and observe a right-skewed distribution of connections over distance, in which connections decrease almost exponentially from the short-range to the long-range. The distance-modulated connectivity found in this work may be extended as guidance in neuromorphic hardware designs to determine the amount of long-range and short-range connections required in the network and the allocation of memory resources in the chips. Specifically, our findings can reduce the number of long-range connections needed for the network, which may decrease the number of memory cells required per neuron to store both source addresses and tag information. This work has the potential to offer a generic model-based memory optimization method in neuromorphic hardware designs.

Acknowledgements

I would like to express my deepest appreciation to Vanessa and Stan for their incredible support throughout the project. I am particularly grateful for their help when I became overwhelmed and got lost. Our weekly brainstorming sessions have built up every brick of this project, and I highly valued their generous advice on how to approach problems we encounter on the road, and their timely feedback on questions that troubled me. I cannot accomplish this project without these two's help. I would like to extend my deepest gratitude to Prof. Giacomo Indiveri, who offered the opportunity to make this project happen and brought Vanessa, Stan and I together. I'm also thankful for Giacomo's insightful suggestions and constructive advice on the general direction of the project. His encouragement always helps me to regain confidence when I have doubts in the work.

I am also extremely grateful to Renate Krause for her precious inputs on finding a task to test the RNN model and inspirational discussions and advice. I owe Chenxi Wu many thanks for his valuable suggestions on the neuromorphic hardware implementation part and extensive discussions on future implications of the current work. Special thanks to Dr. Charlotte Frenkel for her expertise in MorphIC and invaluable insights on neuromorphic routing schemes. And I must thank Dr. Florian Meier, who has inspired me of the initial idea of the project.

Finally, I would like to my families who have always been by my side throughout the ups and downs. I'd like to show my gratitude to my dearest friends, Hussein, Chen and Sihan, who have been there to offer me mental support, specially to Hussein for his proofreading and technical support. I thank Keethana, Joan, Adrian, Hanna, and Guillem, with whom we created a wonderful atmosphere at home during the stressful pandemic situation. Last but not least, I cannot express enough how thankful I am to Nil, for everything.

Contents

Chapter 1

Introduction

Motivation

Memory constraint represents one of the main problems in the design of computing systems, particularly in neuromorphic hardware designs, which aim to build brain-inspired processing systems. The human brain consists of around 100 billion neurons, and each neuron can have 1000 to 10'000 connections [Akopyan et al., 2015]. The biological brain is characterized by dense local connections, which may be a product of natural selection to minimize resource-usage and optimize the efficiency of signal transmission. Thus, understanding how connectivity works in the brain is crucial for finding an efficient solution to memory allocation and optimization in neuromorphic hardware implementations.

1.1 Small world theory: connectivity and geometric distance

The human brain weighs 2% of the body-weight but consumes approximately 20% of the resting metabolic energy [Holliday et al., 1967]. With such a high energy demand, natural evolution drives the biological neural system towards minimal energy consumption while maintaining all essential functions. Biophysical properties, such as the size and the number of neurons existing in the brain, achieve a trade-off between the minimal metabolic energy required and the complexity necessary for generating new behaviors for survival purposes [Laughlin and Sejnowski, 2003, Hasenstaub et al., 2010, Sengupta et al., 2013, Remme et al., 2018, Gollo et al., 2018]. The wiring strategy in the biological nervous system is another crucial factor for achieving efficiency because neural information processing is the most energy-expensive cortical activity and the biological nervous system prefers the shortest possible wiring strategy to minimize the energy used during signal transmission [Laughlin, 2001, Harris and Attwell, 2012, Harris et al., 2012]. Empirical studies have shown that the positioning of neurons and synapses follows a wiring principle, which produces sufficient complexity in the network at a minimum energy expenditure [Bullmore and Sporns, 2012, Wang and Clandinin, 2016, Gollo et al., 2018].

This efficient wiring strategy is hypothesized to follow a *small-world* topology: most connections in the brain are assembled into local circuits while long-distance connections can still occur although the global connections prefer the shortest wiring diagram [Bullmore and Sporns, 2012, Sporns and Zwi, 2004, Watts and Strogatz, 1998, Gastner and Ódor, 2016, Cherniak, 1994, Watts and Strogatz, 1998]. The small world hypothesis has gained significant attention as a graph theory approach to understand the biological neural system [Bassett and Bullmore, 2006, 2017, Bullmore and Sporns, 2012]. The small-worldness has been observed in cross-species empirical studies [Watts and Strogatz, 1998, Sporns and Zwi, 2004, Alexander-Bloch et al., 2013]. The small-world topology in the brain allows optimal information processing: it can form locally clustered connections and global integration where a few long-range connections will be employed so the brain can achieve an efficient information transmission at a low wiring cost [Bassett and Bullmore, 2006, 2017]. This theory highlights the neural connectivity's dependence on geometric distance. Perinelli et al. [2019] demonstrate that the distance constraint may serve as an intrinsic governing factor for the wiring architecture. Thus, investigating the role of geometric distance in the sense of hardware implementation may be a potential approach to study the connectivity in neuromorphic devices and ultimately provide a solution to reducing the memory requirement.

1.2 Memory reduction in neuromorphic hardware

Inspired by the small-world characteristics—dense local clusters and long-range information transmission, DYNAP-SE [Moradi et al., 2017] adopts a mixed routing strategy where neurons are divided into clusters and follow a two-phase tag-based hierarchical routing scheme. The first stage of the routing scheme uses a point-to-point strategy to route neurons to a subset of intermediate neurons. The intermediate neurons broadcast the tag within the local cluster. The tags are stored in content addressable memory, representing the target neurons used in phase 2, such that the set-up spares the memory required for saving the tag information. DYNAP-SE manages to reduce memory requirement due to the densely clustered structure and a hierarchical routing scheme. However, DYNAP-SE can only reduce memory usage for storing address tag IDs, but still requires significant memory resources to store full connectivity information [Moradi et al., 2017, Frenkel et al., 2019].

Another neuromorphic processor following small-world brain topology is MorphIC [Frenkel et al., 2019]. MorphIC employs different types of routers at each level, and the combinations of mixed routing schemes produce high efficiency. In MorphIC, neurons from the same core communicate via a crossbar operation. At the inter-core and intra-chip level, neurons also follow a crossbar approach. The only change from communication at this level is that the connectivity requires 3 bits per neuron to specify the mapping onto the other three cores. The inter-chip communication uses a point-to-point routing scheme, and this level is the most memory-expensive part, for it requires enough memory to store information for inter-core and inter-chip connectivity. The crossbar operation in MorphIC largely reduces memory requirement. However, this all-to-all connectivity also results in a drawback that free resources cannot be reallocated, leading to inefficiency when implementing a sparse neural network.

Although the existing approaches inspired by small-world brain topology [Moradi et al., 2017, Frenkel et al., 2019] have shown impressive results in memory reduction, none has, nevertheless, explored the dependence of connectivity on geometric distance. Moreover, existing approaches to memory reduction are mostly hardware-based [Young et al., 2019], and a directly drawn solution from biological neural network models that are implemented into the hardware device is still missing.

1.3 Bringing the biological solution into ANN models

Our current work fills in the two missing dots mentioned above: a solution that reflects the distance-dependent connectivity in the small-world network topology and is directly drawn from neural network models. Inspired by biological nervous systems, we propose a generic implementation of distance constraint in artificial neural network (ANN) models, following the small-world network theory. We implemented the distance constraint in a fully-connected neural network (FCNN) for MNIST Handwritten Digit Classification [LeCun and Cortes, 2010] and in a recurrent neural network (RNN) for a 3-bit memory task [Sussillo and Barak, 2013]. The distance constraint is used during training to regularize the model's loss such that weights associated with longer distances get more substantial penalties. We compare our distance-constrained models with models using L^p regularization, which only penalizes weights based on the magnitude of the weight, to demonstrate the impacts of the distance constraint. Both L^p-regularized and distance-constrained models are evaluated based on the model's performance and weight distribution with, and compared with a baseline model using no constraint. Our results show that although both types of constrained models achieve comparable performances on the MNIST and the 3-bit memory tasks, the distance constraint pushes the network's weights towards short-distance ranges, whereas L^p regularization does not show a distance-dependent preference. We then apply a dynamic pruning method to sparsify constrained models till the model can no longer achieve a bottom-line accuracy. For sparse models, we compare the maximum sparsity achieved under both constraints and the distribution of remaining connections over distances. We found that both types of constrained models reach a similar level of sparsity. However, connections drop exponentially from short-range to long-range distances distance-constrained models, while L^p-regularized models show no preference over ranges of distances. Our work suggests distance-dependent connectivity in distance-constrained ANN models, which indicate a potential network-based strategy to position neurons in a neural network. We extend the discussion of the current work to potential implications in neuromorphic hardware for reducing memory resources in their routers and network design.

Chapter 2

Methods

In this chapter, we introduce and detail the proposed distance constraint and its implementation. Then, we describe how we train an FCNN model for MNIST classification task [LeCun and Cortes, 2010] and an RNN model for a 3-bit memory task [Sussillo and Barak, 2013]. Next, we present a dynamic binary-search pruning method to obtain the sparsest topological structure of models. Finally, we discuss the metrics we used for evaluating two types of constrained models, including the model's connectivity and sparsity, as well as the performance.

The experiments on this thesis were carried out in simulated layouts, which do not require any actual hardware implementation. The neural networks defined in this work were built using the *PyTorch* library [Paszke et al., 2019]. The source code for the distance constraint definition, how to run described experiments, and the script to produce figures shown in the thesis can be found here:
`https://code.ini.uzh.ch/xyao/energy_constrained_model/tree/master`.
The experiments from this thesis were run on a machine with graphics acceleration[1]. The complete list of the packages needed and information about how to run the code can be found in the repository.

2.1 Definition of the distance constraint

The concept of *distance* used in this thesis stands for the geometric distance between different units in the neural network (herein also referred to as axonal distance or connection distance). The *distance constraint* is defined as a distance-modulated penalty, which gives more penalties to weights associated with long distances and is added as a regularization term to the loss function during training.

[1] PyTorch version: 1.6.0, Cuda version: 10.1

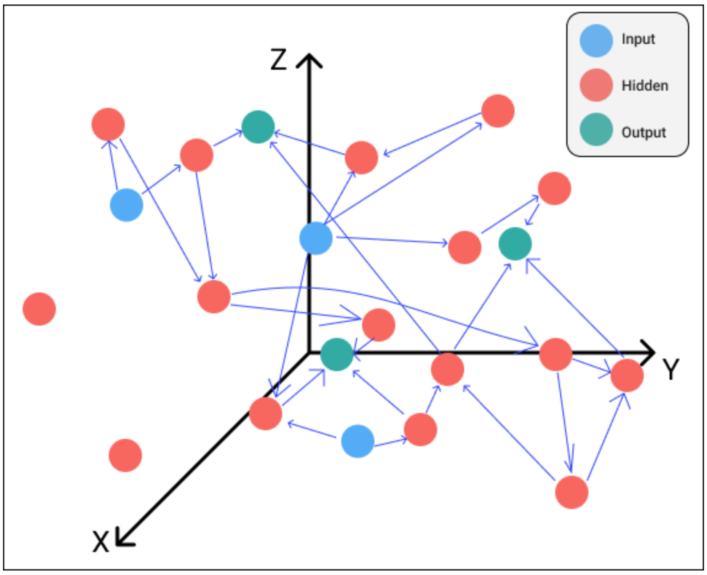


Figure 2.1: A visual representation of the network connectivity in 3-d space.

2.1.1 Represent spatial distance in a neural network

To represent the distance information, we generate a 3D synthetic point cloud from a uniform distribution and assign the coordinates of each point to a neuron in the network to represent the location of the neuron. Figure 2.1 illustrates the visual representation of the distance-constrained network, in which most connections are local while long-distance connections can occur.

We compute the p-norm distance between any neuronal pair and store the distance information in a distance matrix. The distance matrix is normalized such that the average distance between any two nodes is 1. We map the distances to the corresponding weights by applying any required shape transformation of the matrix (Figure 2.2).

2.1.2 Implementing the distance constraint

We perform an element-wise multiplication for the distance and weight matrices and take the L^p-norm of the product. We then implement the matrix product into the loss function an L^p regularization such that connections

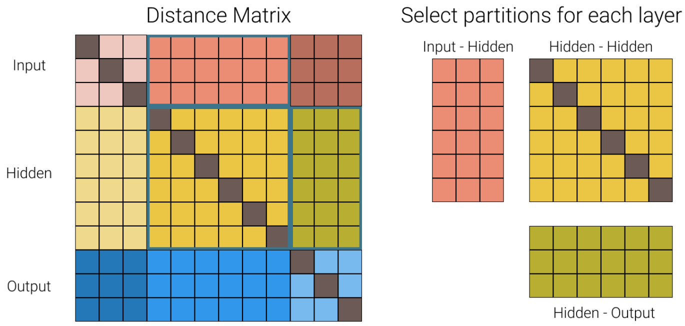


Figure 2.2: We divide the distance matrix into partitions to match for the shape of their corresponding partitions in the weight matrix. To perform an element-wise multiplication, the distance matrix is transposed to match the shape of the distance matrix.

with a long-distance get a stronger penalty.

To control the strength of constraint implemented into the network, we introduce a scaling hyper-parameter α. We tune the value of α to balance the strength of the constraint and the performance of the model. The mathematical description for the implementation of the distance constraint is shown as below:

$$\mathcal{L} = \mathcal{L}_{acc}(y, \hat{y}) + \alpha \sum ||w_i \cdot d_i||^p,\ \alpha \leq 1 \tag{2.1}$$

where w_i and d_i denote the i-th entry in the weight matrix and the distance matrix, respectively.

2.2 Network architecture and datasets

As a sanity check, we compare L^p-regularized models with distance-constrained models to verify the impacts of distance information on networks' behaviors. A baseline model where α equals zero is shown for both constrained cases.

2.2.1 Distance-constrained FCNN on MNIST dataset

We test the proposed distance-constrained regularizer on the MNIST handwritten digit dataset [LeCun and Cortes, 2010]. The training error is computed via the negative log-likelihood loss function. Experiments are repeated ten times using ten different seeds to control the randomness. We report the performance of the model as the inferring accuracy on the test split of the dataset. We use a two-layer FCNN for this task; it has 784 input units, two hidden layers with 500 neurons in each, and ten outputs. The network weights are initialized using Kaiming Uniform initializer [He et al., 2015].

2.2.2 Distance-constrained RNN on a 3-bit memory task

We use a 3-bit memory task [Sussillo and Barak, 2013] to evaluate distance-constrained RNN models. As described in Sussillo and Barak [2013], the goal is to train an RNN model to solve tasks by creating fixed points and using the fixed point dynamics to solve the memory task. During training, the RNN model receives three independent sequences of binary inputs, consisting of -1, 0, and +1, which set each channel's state (or ”bit”). The model is trained to recall the last non-zero input to the channel and to ignore inputs from the other two channels (see Figure 3.4 and Figure A.3 for a visual representation of the task). The trained model will create eight fixed points representing network activation states where the network is stable and no longer moves around. The model can solve tasks relying on the fixed point dynamics to transition from one state to another. Thus, identifying the trained model's fixed points is an important metric to confirm if the constrained model can perform adequately.

To observe the fixed points, we use principal component analysis (PCA) on the hidden states of the network, as suggested in [Golub and Sussillo, 2018]. The well-trained RNN model should display cubic-like PCA trajectories, where eight fixed points can be identified (see Figure A.2). Apart from the dynamics of the trained network, we evaluate the network's performance by computing the mean squared error (MSE) between the model's estimates and true targets. We repeated the training with 20 manual seeds (n=20) and reported the accuracy as the median of MSE obtained from 20 experiments for each α value. The RNN architecture we used here is adapted from Golub and Sussillo [2018]. In the RNN, there are three input units and three output units corresponding to the three memory bits and one hidden layer with 64 hidden neurons. The network weights and biases are initialized uniformly.

2.3 A dynamic binary-search pruning algorithm

We develop a pruning method based on the binary-search algorithm to find the sparsest topology the model can achieve while all models satisfy some minimum accuracy requirement (Θ_a). This method is used in both FCNN and RNN models when analyzing the sparsity of the network topology.

To ensure all sparse network models can maintain a comparable level of performance, we determine an accuracy threshold (Θ_a) based on the performance of the dense models of the same kind, i.e., either FCNN or RNN.

This accuracy threshold is used to adjust the weight threshold for pruning. When initializing the model, we compute the variance (s_w) of the initial weights and define the weight threshold (Θ_w) as s_w divided by a scalar δ. During training, the value of δ is adjusted concerning the pruned model's performance, whereas the variance of the initial weights stays the same once initialized.

At every training iteration, all weights smaller than the threshold Θ_w are removed from the network, and the model's performance is evaluated against an accuracy threshold Θ_a for every 20 epochs. If the model performs better than the threshold, we store the checkpoint of the current model and continue training with a larger threshold Θ_w such that more weights can be removed; otherwise, we restart the model from the last checkpoint and reduce the threshold Θ_w to continue training. The weight threshold Θ_w is adjusted based on the model's performance. We apply a binary-search algorithm to determine the value of Θ_w, and we stop the searching if one half of the search interval is smaller than the arbitrary stopping criterion (δ_s). algorithm 1 demonstrates the dynamic binary-search pruning algorithm formally.

2.4 Metrics for evaluating constrained models

2.4.1 Representing weight distribution against distances

We plot the weight distribution against the internodal distance using a Hexbin plot. In each plot, we fix the range of the x-axis (internodal distance) and show the log scale distributions of weights at each distance range. In sparse networks, we remove zero-weights and their associated distances such that the weight distribution only represents connections in the network.

Komogrov-Smirnov test for statistical signifcance In addition to the Hexbin plots, we statistically demonstrate the difference in weight distributions of the two types of constrained models with a two-sample Kolmogorov-Smirnov (KS) test [Massey Jr, 1951] and report the results using a cumulative density function (CDF). We report D-statistics and p-value to determine significant differences between the weight distributions from two types of constrained models.

For the RNN models, we do a statistical analysis on the shape of the weight distributions around the median of the distance. Therefore, we separate the weights of those distances into two groups: weights that are under the median (i.e., left tail) and those over the median (i.e., right tail). We perform KS-test on both groups and report the result as a CDF.

Algorithm 1: A dynamic binary-search pruning algorithm.

```
Result: A model with the sparsest connectivity it can achieve while
        meeting the performance criterion ($\Theta_a$)
initialization;
weight threshold is $\Theta_w = s_w/\delta$;
$\hat{\delta}$ denotes the last saved $\delta$ value (or $2\delta$ if there is no saved
  checkpoint);
while $\delta > \delta_s$ do
    for i in max(epoch) do
        training session;
        remove weights smaller than $\Theta_w$;
        if model accuracy is smaller than $\Theta_a$ then
            if loss has not converged then
                continue
            else
                set a new scalar $\delta = \frac{(\delta+\hat{\delta})}{2}$ and a new threshold
                  $\Theta_w = s_w/\delta$;
                load the last checkpoint and continue the training
                  session with the new threshold;
            end
        else
            save the model;
            set a new scalar $\delta = \frac{(\hat{\delta}-\delta)}{2}$ and a new threshold
              $\Theta_w = s_w/\delta$;
            continue the training session with the new threshold;
        end
    end
end
```

2.4.2 Network sparsity and internodal distances

Network edge density in constrained models We compute the edge density as a ratio of the number of non-zero weights to the total number of weights in the network. We report the result as the median edge density as well as the standard deviation from n experiments for each α value tested.

Connectivity distributions over distances To analyze the sparse topology of the network, we represent the connection distribution over distances in histogram plots. The network connection is determined by counting the number of non-zero weights within each distance range. We removed the weights associated with only self-loops when computing the network connection. Thus, the distribution of network connectivity represents only "meaningful" connections at each range of distance.

2.4.3 Models' performance evaluation

Apart from the evaluating method in Chapter 2.2.1 and Chapter 2.2.2, we also perform a KS test on the accuracy to evaluate if the performance of p-norm distance constrained models and L^p regularized models are significantly different. A p-value and D-statistics are reported to determine the significance.

Chapter 3

Results

This chapter describes the key findings of the current work. First, as a proof-of-concept, we demonstrate results from a distance-constrained FCNN model on MNIST dataset [LeCun and Cortes, 2010]. We show that the distance-constrained regularization compresses the network's weights into a shorter distance range than L^2 regularization. We then present the same analysis on an RNN model using a 3-bit memory task [Sussillo and Barak, 2013] and observe similar weight vs. distance distribution. In the last section, we discuss network behaviors after pruning and report that the distance constraint pushes the network weights towards a right-skewed distribution.

3.1 Distance-constrained FCNN models

For the sake of simplicity, we started with a shallow-layered FCNN model on MNIST [LeCun and Cortes, 2010] as described in Chapter 2.2.1. The implementation details of the distance constraint can be found in Chapter 2.1.2.

The distance constraint compresses weights into shorter distance range

We trained both types of constrained models for different constraint levels, including the cases where there is no constraint. We show the weight distribution over internodal distance at the early stage (the first epoch) and the end of the training for different constraint levels, which is represented by the value of α for both constrained models (Figure 3.1). We observe that in baseline models (top panels in Figure 3.1.a and b), weights are randomly distributed over internodal distance. The weight distribution in the L^2 regularized model has a higher amount of smaller weights than the distribution seen in the baseline model (Figure 3.1.a). However, the weight distribution pattern over distances in L^2 constrained models is similar to the baseline model—weights are randomly distributed over distances. In distance-constrained models, we observe that the distance constraint not

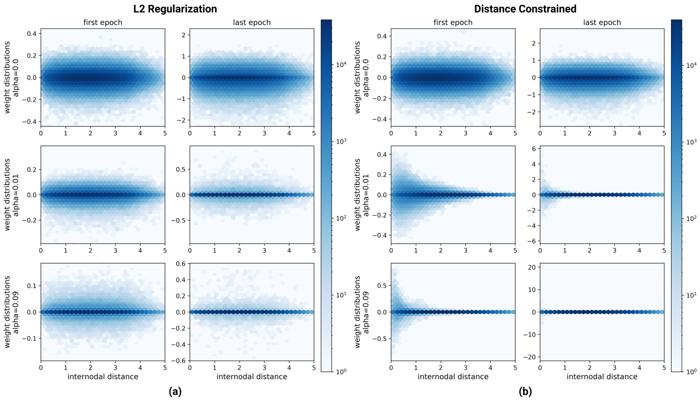


Figure 3.1: An example of weight distribution over internodal distances (log scale) for $\alpha \in \{0.0, 0.01, 0.09\}$, where $\alpha = 0.0$ indicates the baseline model and $\alpha = 0.01$ and $\alpha = 0.09$ represent the weakest and the strongest constraint respectively.

only pushes most weights into small values but compresses the weight distribution into a shorter internodal distance range, which can be seen after only one epoch of training. The compressing effect of the distance constraint on the weight distribution is increasing with the strength of the constraint, i.e., with increasing α values.

Comparable performance in two types of constrained models

We evaluate the constrained models' accuracy to check whether the different weight distributions lead to changes in the model's performance. Figure 3.2 shows that test accuracies of both types of constrained models are decreasing with increasing strength of constraint, i.e., increasing α values. L^2 regularized models weakly outperform the distance-constrained models for relatively weaker constraints, i.e., for $\alpha \in \{0.01, 0.02, 0.03\}$, $0.001 < p < 0.05$ and $D > 0.6$. When posing stronger constraining effects, i.e. larger α values ($\alpha > 0.04$) the distance-constrained models achieve better performance than L^2; the distance-constrained models strongly outperform L^2 models for $\alpha \in \{0.07, 0.08, 0.09\}$ ($p < 0.001$, $d > 0.6$). Models using 1-norm distance constraint and L^1 regularization cannot obtain convergence in loss, for the constraint strength is too big. Thus, results from the 1-norm constrained models are not shown.

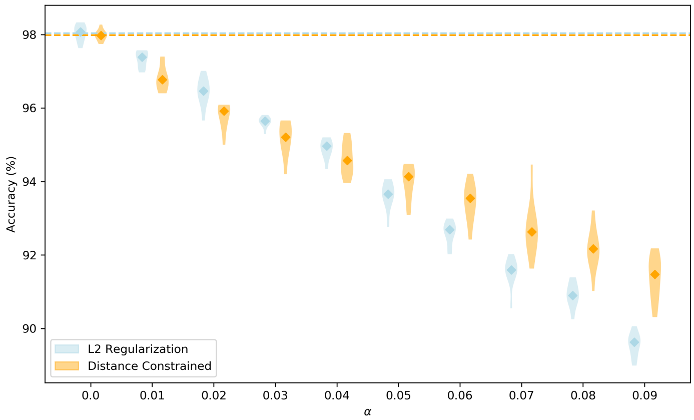


Figure 3.2: Inference accuracy (in percentage) of L^2- and 2-norm distance-constrained models on the test split of the MNIST dataset from 10 experiments (n=10). The diamond-shaped marker in each violin-shaped distribution stands for the median of the model's accuracy. The shape of the violin represents the distribution of the accuracy of each experiment. The accuracy of baseline models is marked by the horizontal dash line.

3.2 Distance constrained RNN models on a 3-bit memory task

Although the implementation of brain-inspired distance constraint in FCNN models on MNIST task provides a start for us to proceed, the FCNN operates in a fundamentally different way from the biological neural network. The MNIST dataset [LeCun and Cortes, 2010] is quite different from cognitive tasks that happen in the biological brain. Since what we are interested in is whether the distance constraint in an ANN model can display similar behaviors as observed in a small-world brain network, the RNN model is more attractive as it shares fundamental similarities to the biological brain. We chose a 3-bit memory task [Sussillo and Barak, 2013] as our dataset because this task simulates how memories are represented in the biological brain (see Chapter 2.2.2 for detailed task description).

3.2.1 Evaluating the performance of constrained models

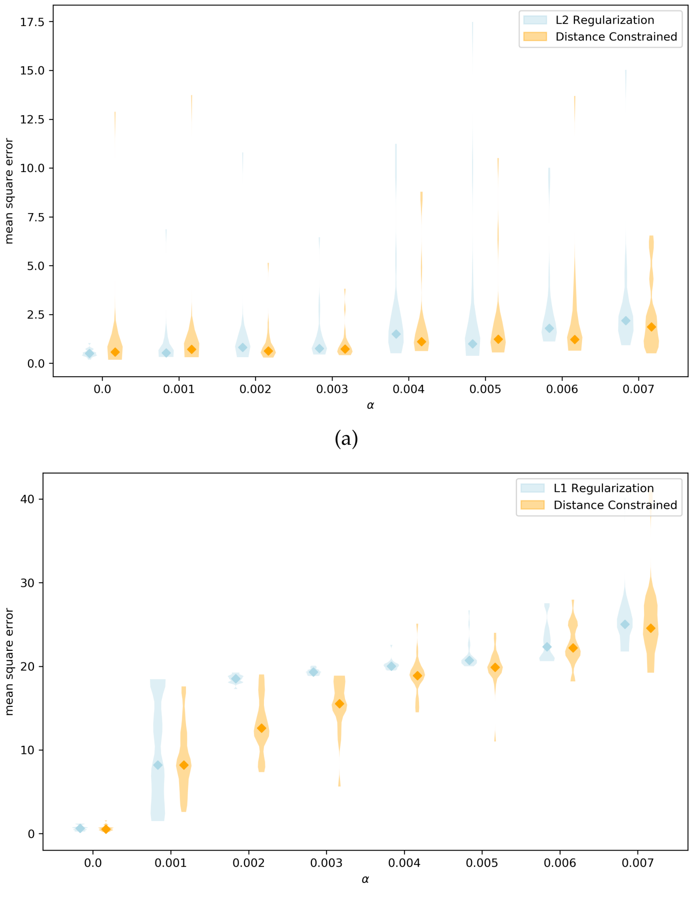
In the 3-bit memory task [Sussillo and Barak, 2013, Sussillo, 2014, Golub and Sussillo, 2018, Maheswaranathan et al., 2019], one crucial feature is the fixed point dynamics of the trained model. The eight fixed points mark the network activation states at which the model no longer moves around, and the dark edges displayed as trajectories represent the transition probability from one fixed point to another. We use the dynamics of the fixed points as well as the mean squared error of the model's estimation to evaluate the performance of each model.

Dynamics analysis We perform a principal component analysis (PCA) on the activation states of the trained model and present the dynamics of the trained network in Figure A.2 (see Chapter A.2). As shown in Figure A.2, the PCA trajectories of all models, p-norm distance-constrained or L^p regularized, roughly follow a cubic-like trace. At each corner of the cube, we identify a fixed point which marks the stable network activation states (8 in total). This result is consistent with Sussillo and Barak [2013], Sussillo [2014], Golub and Sussillo [2018], Maheswaranathan et al. [2019]. Thus, adding constraints to the RNN model does not affect the network dynamics, and there is no difference observed between models using L^2 regularization and the distant constraint. We ran the same experiments with L^1-norm for both constrained models, and we again observe no difference among models (see Figure A.2).

Inference performance We report the model's performance in the form of MSE (displayed in percentage). As shown in Figure 3.3.a, models regularized by a 2-norm constraint have shown a similar performance as the baseline model ($\alpha = 0.0$), although the accuracy distribution gets more sparse with stronger constraint (when $\alpha \geq 0.004$). However, in 1-norm constrained models, there's a more noticeable decrease in the performance with stronger constraint. When comparing the MSE of *2*-norm constrained models with those using *1*-norm, the former has an overall better performance than the latter (Figure 3.3). In Figure 3.4, we represent the models' performance in a visualized way as a cross-check. In 2-norm constrained models, the predicted outputs of the input (Figure 3.4 'blue') overlays well with the true outputs (Figure 3.4 'yellow'). However, in 1-norm constrained cases, more 'blue' lines can be seen, indicating a higher error rate, which is consistent with the MSE plot shown earlier (Figure 3.3). Although there is a deviation from the 2-norm constrained to the 1-norm constrained models' performance, we can nevertheless see a correct model evaluation trend in the MSE plots. Thus, the MSE can be a good indicator for evaluating the model's performance.

3.2.2 Distance constraint weakly modify the weight distributions

We plot the weight distribution over distances for the RNN constrained model (see Chapter A.2 Figure A.4). We show that when using stronger constraints ($\alpha = 0.05, 0.07$), weights in the distance-constrained models are mostly distributed in the shorter range of distances (distance between 0 to



(a)

(b)

Figure 3.3: Mean squared error plots across n=20 experiments. In each violin-shaped plot, the median of the MSE at each α value is marked by a diamond-shaped marker. (a) L^2 regularized models vs. 2-norm distance-constrained models. (b) L^1 regularized models vs. 1-norm distance-constrained models. The baseline models are denoted as models at $\alpha = 0.0$, and the strength of the constraint increases with α.

1) compared with a random distribution of weights in L^p-regularized mod-

3. Results

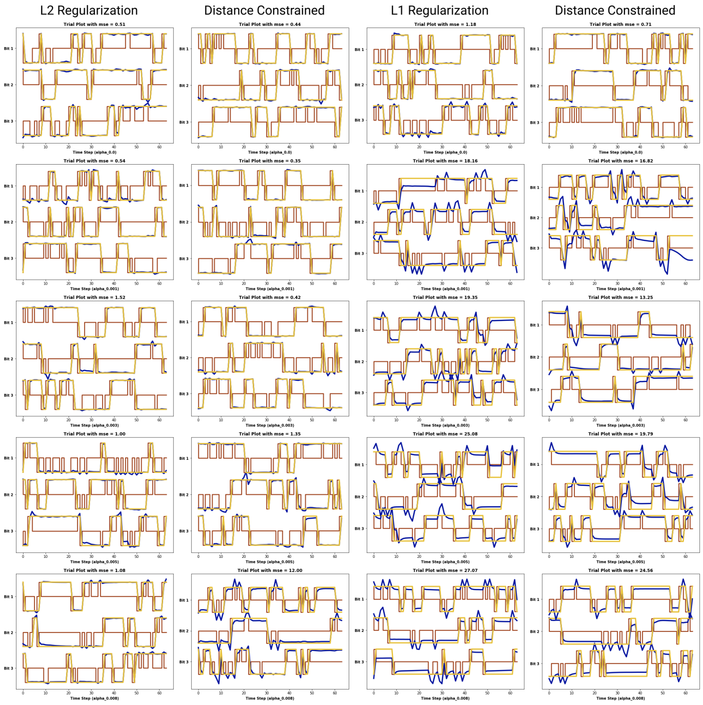


Figure 3.4: Example trial to visualize the performance of constrained models on the 3-bit memory task. The performances with the computed MSE of models using L^2-regularization, 2-norm distance-constraint, L^1-regularization, and 1-norm distance constraint are displayed from the left-most column to the right-most column respectively. In each plot, red lines indicate binary input sequences consisting of -1, 0, and $+1$, yellow lines represent target outcomes of the model which are either -1 or $+1$ and the blue lines stand for estimated outputs of the model which should ideally follow the same pattern as in yellow lines. From the top panels to the bottom ones, the strength of the constraints increases from none ($\alpha = 0.0$) to the strongest ($\alpha = 0.008$).

els. However, in models using relatively small α values, the trend cannot be observed easily. Thus, we did a statistical analysis to compare the weight distribution in both types of constrained models. We divided the distribution of the weights into two groups—one associated with a shorter distance and the other longer (separated by the median of the distance)—and evaluated any significant difference in the distributions. In Figure A.5 (see Chapter A.2), we show that across all degrees of constraining effect, L^p-regularized models show no significant changes in the weight distribution associated with longer or shorter distances. However, weight distributions in distance-constrained models present a strong preference (with $p < 0.001$) over a shorter distance range (Figure A.5, denoted by 'blue' lines). Thus, distance-constrained RNN models still have some impacts on the weight distribution, albeit rather weak when compared with the compressing effect observed in the FCNN models (Figure 3.1). This may occur because we applied a 10-times weaker scaling parameter (α) in the RNN models than in the FCNN models.

3.3 Constrained sparse RNN models

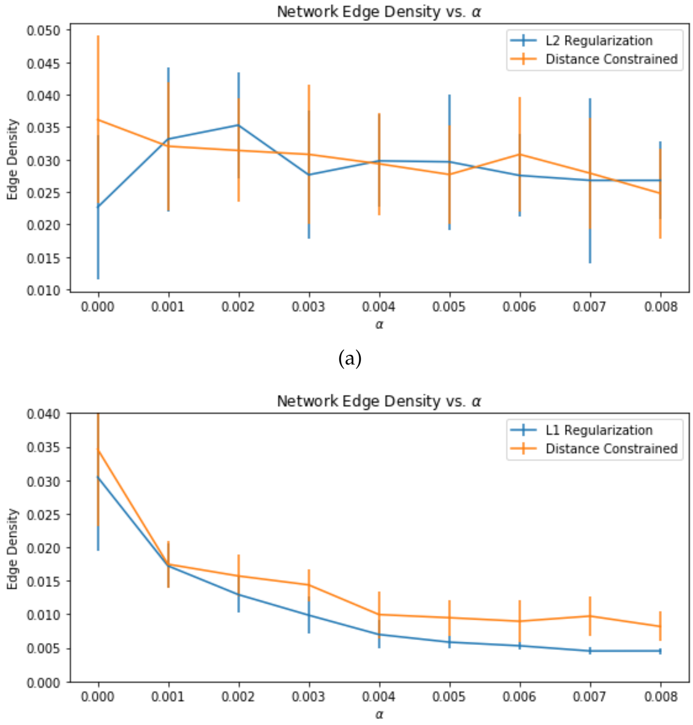
Next, we investigate how the distance constraint affects the sparse connectivity in ANN models. We set a baseline performance for each p-norm constrained model using the MSE reported in Figure 3.3 as a reference. For each pruning experiment, we remove as many weights as possible from the RNN models using a binary-search based pruning method (Chapter 2.3) until the model can no longer reach the baseline performance. Since all saved models have met the baseline performance criterion, we are not showing the accuracy metric in this section, as it is irrelevant.

3.3.1 1-norm constrained models are more sparse than 2-norm

We quantify the constrained RNN models' sparsity as the edge density, which is defined as the ratio of the counts of non-zero weights to all weights. In Figure 3.5, we show that for constrained models using the same p-norm for computation, both types of models (i.e. L^p-regularized or distance-constrained) show a similar level of connection density. However, when comparing 2-norm constrained models (Figure 3.5.a) with 1-norm constrained ones (Figure 3.5.b), we can see that the edge density of 1-norm constrained models is almost half of the density of 2-norm constrained models.

3.3.2 Distance-dependent connectivity distributions

Similar to the previous analysis, we examine the impacts of distance constraint in the sparse RNN models. Since the distance constraint is involved



(a)

(b)

Figure 3.5: Average network edge density (n=20). (a) L^2 regularized models vs. 2-norm distance constrained models. (b) L^1 regularized models vs. 1-norm distance constrained models. The edge density of a network model is computed as the ratio of the total counts of non-zero weights over the total counts of weight in the network.

in modulating the weights from the network, we plot the connection distribution over distance to verify if the network connectivity shows a distance-dependent pattern.

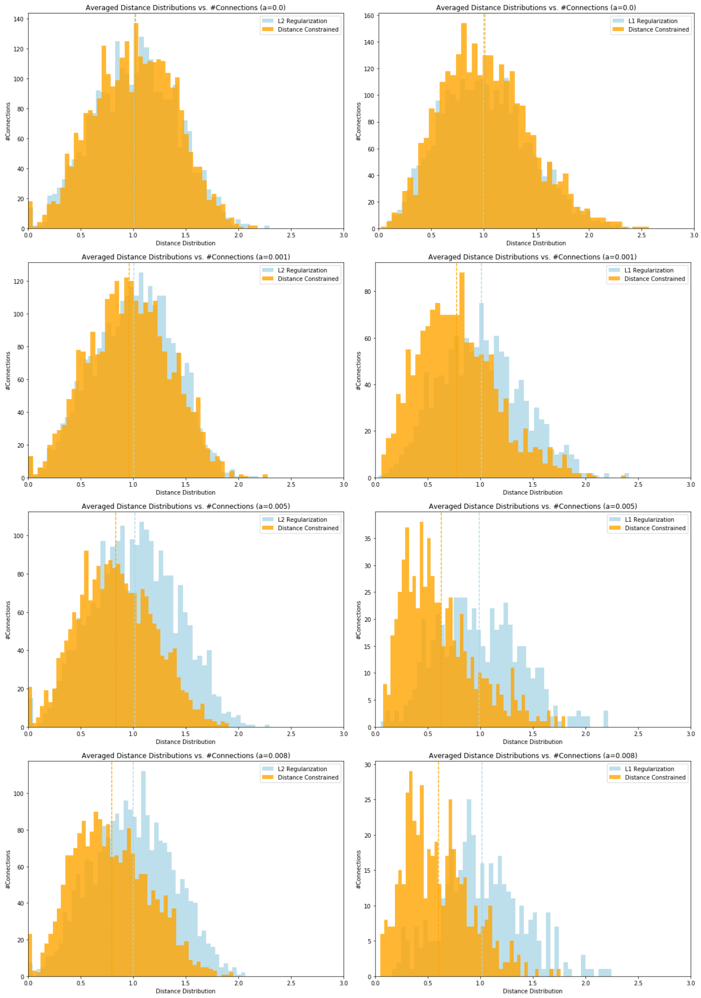
A distance-dependent right skewed connection distribution

In Figure 3.6, we present the connections distribution over internodal distances obtained from the 2-norm (Figure 3.6.a) and the 1-norm (Figure 3.6.b) constrained models for 4 levels of constraints ($\alpha \in \{0.0, 0.001, 0.005, 0.008\}$).

L^p-regularized models ('blue' distribution in Figure 3.6) show a similar distribution of connections as non-constrained models (top row in Figure 3.6). The median of the distance stays about the same across all four different degrees. In distance-constrained models, however, the connections follow a right-skewed distribution and the median of the distance decreases with increasing strength of constraint ('orange' distribution in Figure 3.6). When comparing different p-norm constrained models, 1-norm distance constraint produces a stronger right-skewed connection distribution over distances.

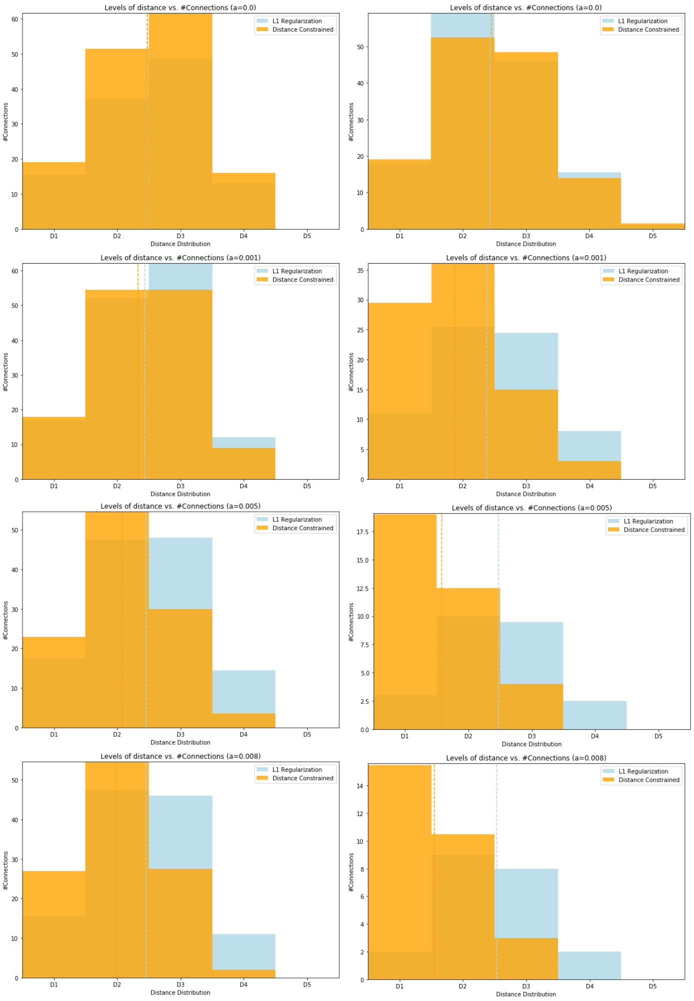
Distance constraint triggers hierarchical connectivity

To show the preference of network connectivity over long- and short-distances, we divided the total range of distance (d=2.5) into five uniform sections and categorized the connections into five ranges of distance—D1 (0.0–0.5), D2 (0.5–1.0), D3 (1.0–1.5), D4 (1.5–2.0) and D5 (2.0–2.5). We define D1–D2 to be short-range, D3 mid-range, and D4–D5 long-range. Figure 3.7 illustrates that in the distance-constrained models ('orange' distribution), the network models show a general preference over short-range connections than long-range, and the connectivity becomes more sparse at longer-range distances. In 1-norm distance-constrained models (Figure 3.7.b), the decrease of connections over distance is exponential. The exponential curve cannot be fitted into the 2-norm distance-constrained models due to the relatively smaller connection distribution at D1 than at D2, despite that the connections drop exponentially from D2 to D3 (especially with larger α values). We perform the same type of pruning and connectivity analysis on the FCNN models for completeness, and we show that in 1-norm constrained FCNN models, the same distance-triggered hierarchical connectivity can be observed as well (see Chapter A.2, Figure A.1).



(a) (b)

Figure 3.6: Counts of network connections distributed over internodal distance, represented as the median from n=20 experiments. (a) L^2 regularized models vs. 2-norm distance constrained models. (b) L^1 regularized models vs. 1-norm distance constrained models. Each bin in the plot represents the counts of connections lying within the corresponding distance range. From top row to the bottom, each row has a corresponding value of α equals to 0.0, 0.001, 0.005, and 0.008.


Levels of distance vs. #Connections (a=0.0)

Levels of distance vs. #Connections (a=0.001)

Levels of distance vs. #Connections (a=0.005)

Levels of distance vs. #Connections (a=0.008)

(a)

(b)

Figure 3.7: Network connections distributed over 5 uniform ranges of distances (n=20). (a) L^2 regularized models vs. 2-norm distance constrained models. (b) L^1 regularized models vs. 1-norm distance constrained models. The value of α from top row to the bottom are 0.0, 0.001, 0.005, and 0.008 respectively

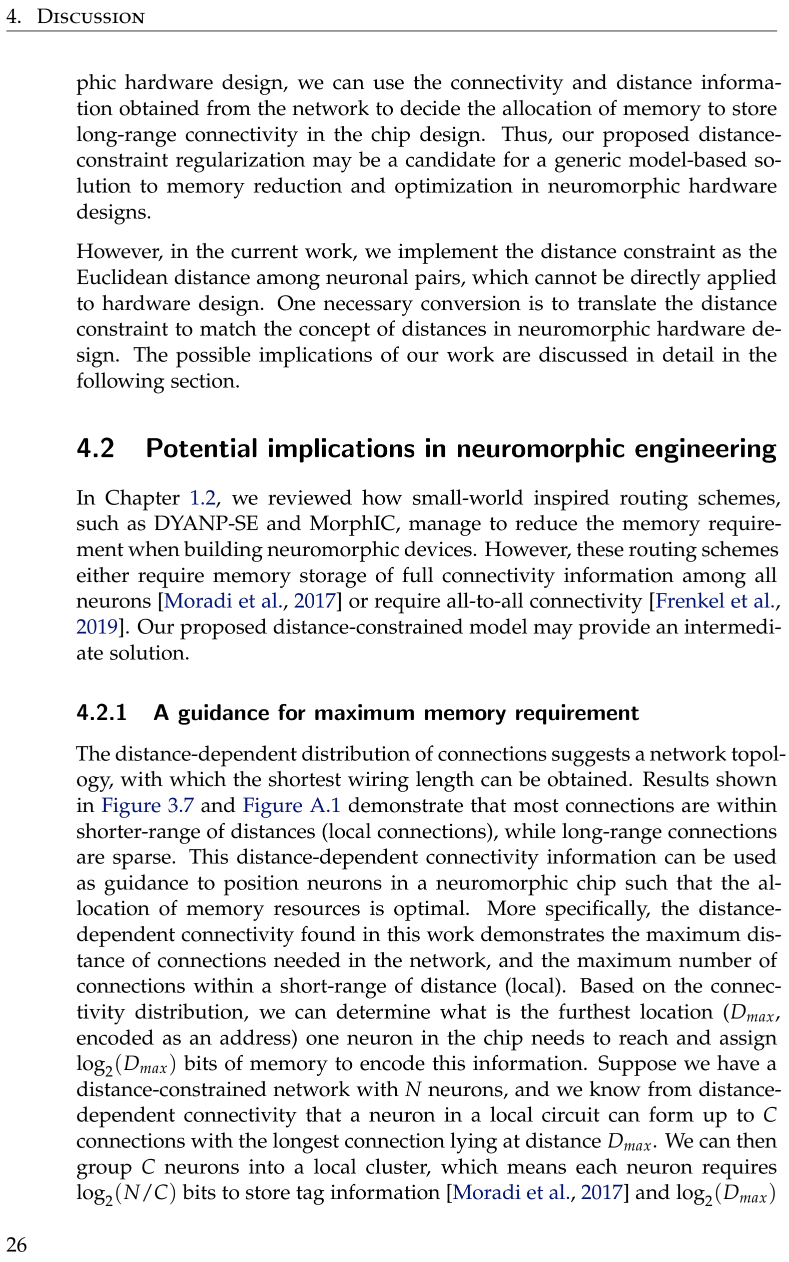
Chapter 4

Discussion

4.1 Small-worldness in distance-constrained models

In this work, we introduce a brain-inspired distance constraint to regularize the ANN models during training. We found that distance-constrained models, regardless of the type of the models, have shown distributions of weights with a preference for short distances. This distance-dependent constraining effect is more prominent in sparsely-connected networks, in which we find a right-skewed connectivity distribution over distances. This preference for short-range (local) connections in network topology is consistent with the small-world brain network. Moreover, we show that we achieve this distance-modulated connectivity without compromising on the network's performance, more than a L^p regularization will do (Figure 3.2 and Figure 3.3). Thus, whenever a small-world topology is needed for an ANN model, one can replace the L^p-regularization with a p-norm distance-constraint.

This model-free distance-constraining effect can be a candidate for a neural-network-model-based strategy to reduce memory requirement in neuromorphic hardware. As shown in Figure 3.7 and Figure A.1, the number of connections located at a shorter range of distances (D1–D2) is exponentially larger than the amount lying in the longer-distance range (D4–D5) in the distance-constrained models. The connectivity distribution seen here indicates a hierarchical preference for distance ranges, i.e., a preference for establishing connections based on the length of internodal distances. Since the distance information is embedded in the trained model, we can use the output topology of the trained model to decide the positioning of neurons such that long-range connections can be minimized. The implication is feasible because, in our model, long-range connections are penalized more, in which only important long-distance ones (neurons that are assigned with a larger value of weight) remain. When translating this observation into neuromor-



phic hardware design, we can use the connectivity and distance information obtained from the network to decide the allocation of memory to store long-range connectivity in the chip design. Thus, our proposed distance-constraint regularization may be a candidate for a generic model-based solution to memory reduction and optimization in neuromorphic hardware designs.

However, in the current work, we implement the distance constraint as the Euclidean distance among neuronal pairs, which cannot be directly applied to hardware design. One necessary conversion is to translate the distance constraint to match the concept of distances in neuromorphic hardware design. The possible implications of our work are discussed in detail in the following section.

4.2 Potential implications in neuromorphic engineering

In Chapter 1.2, we reviewed how small-world inspired routing schemes, such as DYANP-SE and MorphIC, manage to reduce the memory requirement when building neuromorphic devices. However, these routing schemes either require memory storage of full connectivity information among all neurons [Moradi et al., 2017] or require all-to-all connectivity [Frenkel et al., 2019]. Our proposed distance-constrained model may provide an intermediate solution.

4.2.1 A guidance for maximum memory requirement

The distance-dependent distribution of connections suggests a network topology, with which the shortest wiring length can be obtained. Results shown in Figure 3.7 and Figure A.1 demonstrate that most connections are within shorter-range of distances (local connections), while long-range connections are sparse. This distance-dependent connectivity information can be used as guidance to position neurons in a neuromorphic chip such that the allocation of memory resources is optimal. More specifically, the distance-dependent connectivity found in this work demonstrates the maximum distance of connections needed in the network, and the maximum number of connections within a short-range of distance (local). Based on the connectivity distribution, we can determine what is the furthest location (D_{max}, encoded as an address) one neuron in the chip needs to reach and assign $\log_2(D_{max})$ bits of memory to encode this information. Suppose we have a distance-constrained network with N neurons, and we know from distance-dependent connectivity that a neuron in a local circuit can form up to C connections with the longest connection lying at distance D_{max}. We can then group C neurons into a local cluster, which means each neuron requires $\log_2(N/C)$ bits to store tag information [Moradi et al., 2017] and $\log_2(D_{max})$

bits to specify the longest-range of connectivity needed. Thus, the maximum memory (*M)* required by each neuron in the network is:

$$M = \log_2(D_{max}) + \log_2(N/C) \tag{4.1}$$

The memory requirement M in the distance-constrained model is smaller than $\log_2(N)$ because the network is sparse and does not require full connectivity ($D_{max} \leq N$). Thus, we may reduce the memory required for saving neurons' tag information (from N to $\frac{N}{C}$) and for specifying the address ($D_{max} \leq N$).

4.2.2 Directions for future work

Since the distance constraint in this work is based on the Euclidean distance among neurons, we need to convert the distance information into addresses that encodes location information of a spike. However, since this work is based on ANN models and has not tested any spiking neural network, the potential implementation mentioned above is yet to be examined under a more neuromorphic-friendly setup. We propose some potential directions to evaluate the distance-constrained models.

After training with a distance-constraint (see Chapter 2.1.2 for implementation detail), a distance-dependent connectivity topology of the network can be obtained. The distance and connectivity information obtained after training the neural network model can be used to guide the allocation of memory requirements for each neuron in the network in designing neuromorphic devices.

Multiple comparisons can be done to evaluate the efficiency in distance-constrained guided mapping and memory:

1. A distance-constrained model mapped into a neuromorphic chip using the exact location information provided by the network topology.
2. Mapping a distance-constrained model into a chip with shuffled location coordinates, i.e., only distance-modulated connectivity is represented in the model but not the associated distance information.
3. Train a model regularized by a constraint that does not represent distance information and map to the location information provided by the distance-constrained model.

4.3 Related work

Blake et al. [2018] propose a *distance-weight regularization* pruning method to minimize the wiring length in the network while maintaining the accuracy of performances. They demonstrate their results using a fully-connected neural network on the MNIST dataset and show that the distance-weight regularized model tends to have the best performance among comparisons. Blake et al. [2018]'s approach and our current work share some similarity in how the distance-constraint is defined and how this distance-weight regularization can shorten the wiring length in the neural network model, which support the implementation of distance information as a regularization term in ANN models. Blake et al. [2018]'s and the current work, nevertheless, focus on fundamentally different implementation objectives. Blake et al. [2018] aim to shorten the cable length in hardware design using their proposed method, but the concept of wiring length in a neural network model is not applicable in hardware design. However, our goal is to find a small-world topology in the neural network model, which can guide the amount of long- or short-range connections needed in designing a neuromorphic device. The concept of distance in our work is more of a hierarchical distance used in designing the routing scheme rather than a precise location. Thus, despite the similarities mentioned, the ultimate objectives of the two are quite different.

Chapter 5

Conclusion

We presented an implementation of a brain-inspired distance constraint in FCNN and RNN models as a novel approach to trigger distance-dependent connectivity in the ANN models, which follows the small-world theory. We demonstrated that in all distance-constrained models, both the FCNN and the RNN, the distance constraint can, at least to some degree, lead to a weight distribution preferring short distances with minimal loss in performance. In sparse neural networks, the distance constraint can produce a right-skewed distribution of connections. Our observations are consistent with the small world brain topology and reveal the feasibility of representing distance information in ANN models and shorten the wiring length of connections. This discovery of distance-dependent connectivity in ANN models sheds light on developing a model-based solution to modulate the length of connectivity and reduce memory requirement in neuromorphic hardware design.

Our method still requires more modification and verification in different contexts (e.g., spiking neural network, different cognitive-related tasks, and the definition of distances in neuromorphic hardware design. We discuss the potential implications of this neural-network-based strategy to guide the allocations of memory resources in neuromorphic hardware design, which may introduce generic model-based guidance to optimize the memory usage in neuromorphic devices. However, more insights are needed for the application of the current work. Our work provides a new piece of evidence of how understanding the biological neural system and integrating the biological observations into artificial networks may improve the neuromorphic hardware design.

Bibliography

[1] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G.-J. Nam, et al. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE transactions on computer-aided design of integrated circuits and systems*, 34(10):1537–1557, 2015.

[2] A. F. Alexander-Bloch, P. E. Vértes, R. Stidd, F. Lalonde, L. Clasen, J. Rapoport, J. Giedd, E. T. Bullmore, and N. Gogtay. The anatomical distance of functional connections predicts brain network topology in health and schizophrenia. *Cerebral cortex*, 23(1):127–138, 2013.

[3] D. S. Bassett and E. Bullmore. Small-world brain networks. *The neuroscientist*, 12(6):512–523, 2006.

[4] D. S. Bassett and E. T. Bullmore. Small-world brain networks revisited. *The Neuroscientist*, 23(5):499–516, 2017.

[5] C. Blake, L. Wang, G. Castiglione, C. Srinivasa, and M. Brubaker. On learning wire-length efficient neural networks. 2018.

[6] E. Bullmore and O. Sporns. The economy of brain network organization. *Nature Reviews Neuroscience*, 13(5):336–349, 2012.

[7] C. Cherniak. Component placement optimization in the brain. *Journal of Neuroscience*, 14(4):2418–2427, 1994.

[8] C. Frenkel, J.-D. Legat, and D. Bol. Morphic: A 65-nm 738k-synapse/mm ˆ2 quad-core binary-weight digital neuromorphic processor with stochastic spike-driven online learning. *IEEE Transactions on Biomedical Circuits and Systems*, 13(5):999–1010, 2019.

[9] M. T. Gastner and G. Ódor. The topology of large open connectome networks for the human brain. *Scientific reports*, 6:27249, 2016.

[10] L. L. Gollo, J. A. Roberts, V. L. Cropley, M. A. Di Biase, C. Pantelis, A. Zalesky, and M. Breakspear. Fragility and volatility of structural hubs in the human connectome. *Nature neuroscience*, 21(8):1107–1116, 2018.

[11] M. D. Golub and D. Sussillo. Fixedpointfinder: A tensorflow toolbox for identifying and characterizing fixed points in recurrent neural networks. *Journal of Open Source Software*, 3(31):1003, 2018. doi: 10.21105/joss.01003.

[12] J. J. Harris and D. Attwell. The energetics of CNS white matter. *J. Neurosci.*, 2012. ISSN 02706474. doi: 10.1523/JNEUROSCI.3430-11.2012.

[13] J. J. Harris, R. Jolivet, and D. Attwell. Synaptic Energy Use and Supply, 2012. ISSN 08966273.

[14] A. Hasenstaub, S. Otte, E. Callaway, and T. J. Sejnowski. Metabolic cost as a unifying principle governing neuronal biophysics. *Proc. Natl. Acad. Sci. U. S. A.*, 2010. ISSN 00278424. doi: 10.1073/pnas.0914886107.

[15] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, 2015.

[16] M. A. Holliday, D. Potter, A. Jarrah, and S. Bearg. The relation of metabolic rate to body weight and organ size. *Pediatr. Res.*, 1967. ISSN 15300447. doi: 10.1203/00006450-196705000-00005.

[17] S. B. Laughlin. Energy as a constraint on the coding and processing of sensory information, 2001. ISSN 09594388.

[18] S. B. Laughlin and T. J. Sejnowski. Communication in neuronal networks, 2003. ISSN 00368075.

[19] Y. LeCun and C. Cortes. MNIST handwritten digit database. 2010. URL `http://yann.lecun.com/exdb/mnist/`.

[20] N. Maheswaranathan, A. Williams, M. Golub, S. Ganguli, and D. Sussillo. Universality and individuality in neural dynamics across large populations of recurrent networks. In *Advances in neural information processing systems*, pages 15629–15641, 2019.

[21] F. J. Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78, 1951.

[22] S. Moradi, N. Qiao, F. Stefanini, and G. Indiveri. A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (dynaps). *IEEE transactions on biomedical circuits and systems*, 12(1):106–122, 2017.

[23] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[24] A. Perinelli, D. Tabarelli, C. Miniussi, and L. Ricci. Dependence of connectivity on geometric distance in brain networks. *Scientific reports*, 9(1):1–9, 2019.

[25] M. W. Remme, J. Rinzel, and S. Schreiber. Function and energy consumption constrain neuronal biophysics in a canonical computation: Coincidence detection. *PLoS Comput. Biol.*, 2018. ISSN 15537358. doi: 10.1371/journal.pcbi.1006612.

[26] B. Sengupta, A. A. Faisal, S. B. Laughlin, and J. E. Niven. The effect of cell size and channel density on neuronal information encoding and energy efficiency. *J. Cereb. Blood Flow Metab.*, 2013. ISSN 0271678X. doi: 10.1038/jcbfm.2013.103.

[27] O. Sporns and J. D. Zwi. The small world of the cerebral cortex. *Neuroinformatics*, 2(2):145–162, 2004.

[28] D. Sussillo. Neural circuits as computational dynamical systems. *Current opinion in neurobiology*, 25:156–163, 2014.

[29] D. Sussillo and O. Barak. Opening the black box: low-dimensional dynamics in high-dimensional recurrent neural networks. *Neural computation*, 25(3):626–649, 2013.

[30] I. Wang and T. Clandinin. The influence of wiring economy on nervous system evolution. *Current Biology*, 26(20):R1101 – R1108, 2016. ISSN 0960-9822. doi: https://doi.org/10.1016/j.cub.2016.08.053.

[31] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442, 1998.

[32] A. R. Young, M. E. Dean, J. S. Plank, and G. S. Rose. A review of spiking neuromorphic hardware communication systems. *IEEE Access*, 7:135606–135620, 2019.

Appendix A

Appendix

A.1 FCNN supplement figures

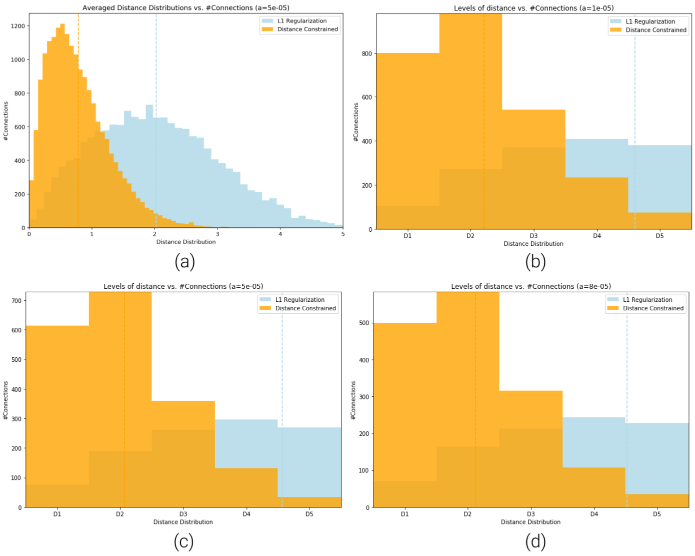


Figure A.1: Connection distribution vs. ranges of distances in sparse FCNN models using 1-norm constraints. (a) A histogram of the medians of connections lying in each distance interval (n=10) with $\alpha = 5e - 5$. (b)–(d) Hierarchical connection distributions over five ranges of distances, with D1–D2 being the shorter ranges of distance (0.0–1.0), D3 the mid-range (1.0–1.5), and D4–D5 the longer-ranger (1.5–2.5).

A.2 3-bit memory task supplement figures

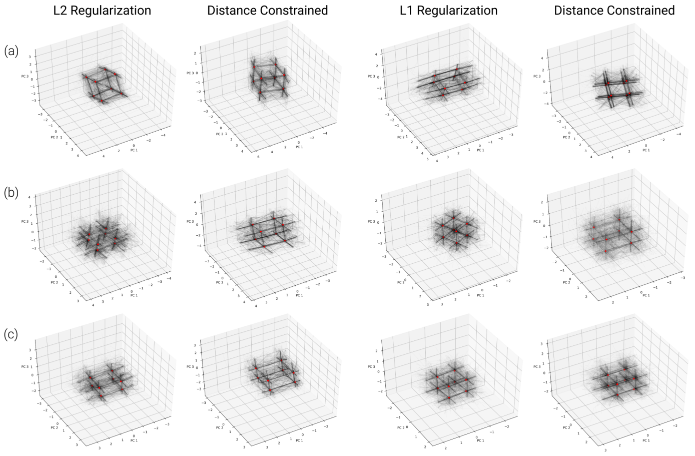


Figure A.2: PCA trajectories of the RNN activation states forL^p-regularized models and distance-constrained models. Row (a)-(c) show the dynamics for models with $\alpha = 0.0, 0.001, 0.005$ respectively. The red dots locating in the corner of the cube mark the fixed points or 8 memory states of the *3*-bit memory task.

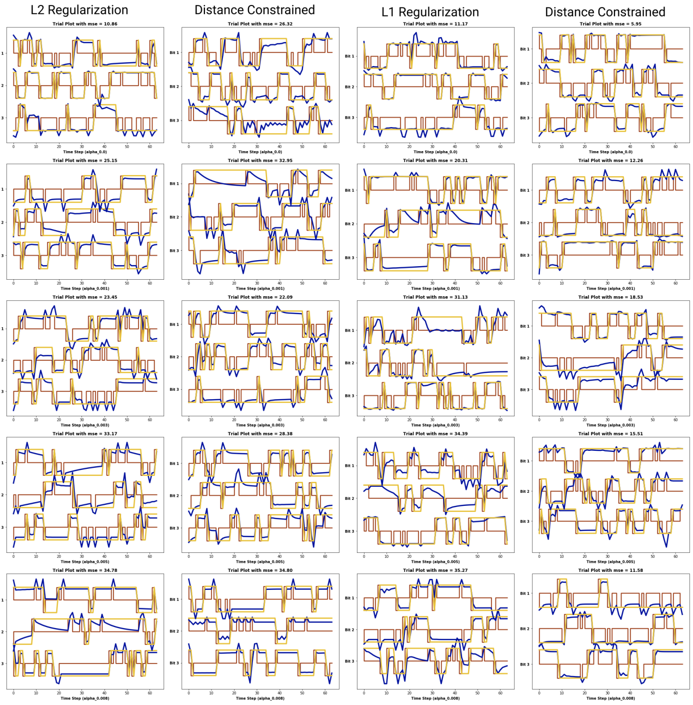


Figure A.3: Example trials to visualize the 3-bit memory task on sparse constrained models.

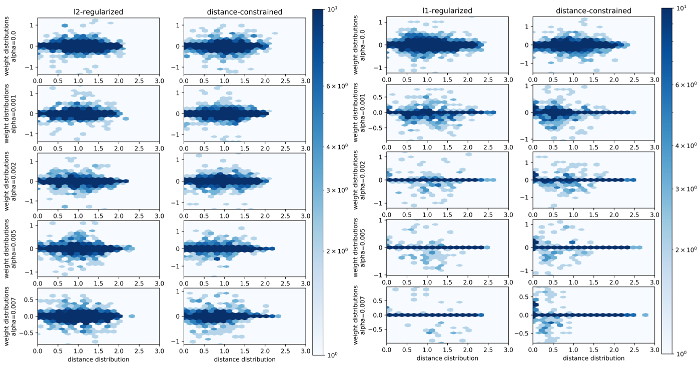


Figure A.4: Weight distribution vs. internodal distances for the constrained RNN models. Weights distribution is displayed in log scale for different α values.

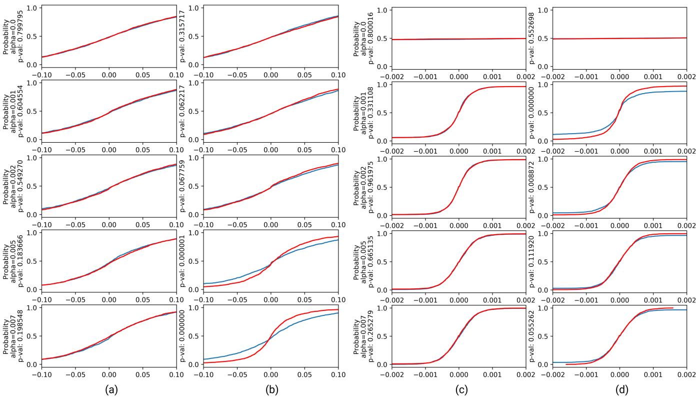


Figure A.5: A cumulative density function demonstrating the statistical analysis on the shape of the weight distribution of RNN models is shown. (a)–(d) represent the cdf computed from models using L^2 regularization, 2-norm distance constraint, L^1 regularization and 1-norm distance constraint. Weights are separated into two groups: the part associated with distances shorter than the median of the distance (denoted by the blue curves), and those with distances longer than the median (denoted by the red curves). In each plot, x-axis denotes the value of weights and y-axis the probability of distribution.

Bibliography

(2016). *NESTML: a modeling language for spiking neurons*. Zenodo.

(2020). Cortexcontrol: A tool for controlling and executing experiments using neuromorphic hardware platforms that communicate through address-event representation. https://gitlab.com/neuroinf/ctxctl_contrib/-/wikis/ctxctl-executables-and-documentation. Unreleased software, Institute of Neuroinformatics, University of Zurich and ETH Zurich.

a Hardware Revolution, B. D. N. (2018). Big data needs a hardware revolution. Nature, 554(7691):145–146.

Abbate, J. (1999). Getting small: a short history of the personal computer. *Proceedings of the IEEE*, 87(9):1695–1698.

Agrawal, B. and Sherwood, T. (2006). Modeling tcam power for next generation network devices. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 120–129. IEEE.

Akopyan, F., Sawada, J., Cassidy, A., Alvarez-Icaza, R., Arthur, J., Merolla, P., Imam, N., Nakamura, Y., Datta, P., Nam, G.-J., Taba, B., Beakes, M., Brezzo, B., Kuang, J. B., Manohar, R., Risk, W. P., Jackson, B., and Modha, D. S. (2015). TrueNorth: Design and tool flow of a 65 mW 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1537–1557.

Amari, S. (1967). A theory of adaptive pattern classifiers. *IEEE Transactions on Electronic Computers*, EC-16(3):299–307.

Amir, A., Datta, P., Risk, W. P., Cassidy, A. S., Kusnitz, J. A., Esser, S. K., Andreopoulos, A., Wong, T. M., Flickner, M., Alvarez-Icaza, R., McQuinn, E., Shaw, B., Pass, N., and Modha, D. S. (2013). Cognitive computing programming paradigm: A corelet language for composing networks of neurosynaptic cores. In *International Joint Conference on Neural Networks (IJCNN)*, pages 1–10. IEEE.

Averbeck, B. B., Latham, P. E., and Pouget, A. (2006). Neural correlations, population coding and computation. *Nature reviews. Neuroscience*, 7(5):358–366.

Backus, J. (1978). Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641.

Balaji, A. and Das, A. (2019). A framework for the analysis of throughput-constraints of snns on neuromorphic hardware. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 193–196. IEEE.

Balaji, A., Das, A., Wu, Y., Huynh, K., Dell'Anna, F. G., Indiveri, G., Krichmar, J. L., Dutt, N. D., Schaafsma, S., and Catthoor, F. (2020a). Mapping spiking neural networks to neuromorphic hardware. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(1):76–86.

Balaji, A., Marty, T., Das, A., and Catthoor, F. (2020b). Run-time mapping of spiking neural networks to neuromorphic hardware. *Journal of Signal Processing Systems*, 92(11):1293–1302.

Barabási, D. L. and Barabási, A.-L. (2020). A genetic model of the connectome. *Neuron*, 105(3):435–445.

Barabási, D. L. and Czégel, D. (2021). Constructing graphs from genetic encodings. *Scientific Reports*, 11(1):1–13.

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.

Bellec, G., Scherr, F., Subramoney, A., Hajek, E., Salaj, D., Legenstein, R., and Maass, W. (2020). A solution to the learning dilemma for recurrent networks of spiking neurons. *Nature Communications*, 11(3625):1–15.

Benjamin, B. V., Gao, P., McQuinn, E., Choudhary, S., Chandrasekaran, A. R., Bussat, J., Alvarez-Icaza, R., Arthur, J., Merolla, P., and Boahen, K. (2014). Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proceedings of the IEEE*, 102(5):699–716.

Binzegger, T., Douglas, R. J., and Martin, K. A. (2005). Cortical architecture. In *International Symposium on Brain, Vision, and Artificial Intelligence*, pages 15–28. Springer.

Bohte, S. M., Kok, J. N., and La Poutré, J. A. (2000). Spikeprop: backpropagation for networks of spiking neurons. In *ESANN*, volume 48, pages 419–424. Bruges.

Bouvier, M., Valentian, A., Mesquida, T., Rummens, F., Reyboz, M., Vianello, E., and Beigne, E. (2019). Spiking neural networks hardware implementations and challenges: A survey. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 15(2):1–35.

Boybat Kara, I. (2020). Multi-memristive synaptic architectures for training neural networks. Technical report, EPFL.

BrainScales (2011–2015). Brain-inspired multiscale computation in neuromorphic hybrid systems (BrainScaleS). FP7 269921 EU Grant.

Bron, C. and Kerbosch, J. (1973). Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Bullmore, E. and Sporns, O. (2009). Complex brain networks: graph theoretical analysis of structural and functional systems. *Nature Reviews Neuroscience*, 10(3):186–198.

Bullmore, E. and Sporns, O. (2012). The economy of brain network organization. *Nature Reviews Neuroscience*, 13(5):336–349.

Burns, R. W. (2004). *Communications: An International History of the Formative Years*, volume 32. IET.

Caporale, N. and Dan, Y. (2008). Spike timing dependent plasticity: A hebbian learning rule. *Annual Review of Neuroscience*, 31(1):25–46.

Carlson, R. and Nemhauser, G. L. (1966). Scheduling to minimize interaction cost. *Operations Research*, 14(1):52–58.

Cazals, F. and Karande, C. (2008). A note on the problem of reporting maximal cliques. *Theoretical computer science*, 407(1-3):564–568.

Ceruzzi, P. E. (2003). *A history of modern computing*. MIT press.

Chen, G. K., Kumar, R., Sumbul, H. E., Knag, P. C., and Krishnamurthy, R. K. (2019). A 4096-neuron 1M-synapse 3.8-pJ/SOP spiking neural network with on-chip STDP learning and sparse weights in 10-nm FinFET CMOS. *IEEE Journal of Solid-State Circuits*, 54(4):992–1002.

Chen, Y., Huang, A., Wang, Z., Antonoglou, I., Schrittwieser, J., Silver, D., and de Freitas, N. (2018). Bayesian optimization in alphago. *arXiv preprint arXiv:1812.06855*.

Chicca, E. and Indiveri, G. (2020). A recipe for creating ideal hybrid memristive-CMOS neuromorphic processing systems. *Applied Physics Letters*, 116(12):120501.

Chowdhury, S. N. and Shah, S. (2022). Hardware aware modeling of mixed-signal spiking neural network. In *2022 20th IEEE Interregional NEWCAS Conference (NEWCAS)*, pages 104–108.

Clarke, D. D. and Sokoloff, L. (1999). *Circulation and energy metabolism in the brain*, chapter 31, pages 637–669. Lippincot - Raven; Sixth Edition.

Covi, E., Donati, E., Liang, X., Kappel, D., Heidari, H., Payvand, M., and Wang, W. (2021). Adaptive extreme edge computing for wearable devices. *Frontiers in Neuroscience*, 15.

Daneshtalab, M., Ebrahimi, M., Liljeberg, P., Plosila, J., and Tenhunen, H. (2010). A low-latency and memory-efficient on-chip network. In *2010 Fourth ACM/IEEE International Symposium on Networks-on-Chip*, pages 99–106. IEEE.

Darwish, T. and Bayoumi, M. (2005). Trends in low-power vlsi design. *The Electrical Engineering Handbook*, pages 263–280.

Das, A., Wu, Y., Huynh, K., Dell'Anna, F., Catthoor, F., and Schaafsma, S. (2018). Mapping of local and global synapses on spiking neuromorphic hardware. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1217–1222. IEEE.

Davies, M., Srinivasa, N., Lin, T.-H., Chinya, G., Cao, Y., Choday, S. H., Dimou, G., Joshi, P., Imam, N., Jain, S., Liao, Y., Lin, C.-K., Lines, A., Liu, R., Mathaikutty, D., McCoy, S., Paul, A., Tse, J., Venkataramanan, G., Weng, Y.-H., Wild, A., Yang, Y., and Wang, H. (2018). Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99.

Davison, A., Hines, M., and Muller, E. (2009). Trends in programming languages for neuroscience simulations. *Frontiers in Neuroscience*, 3:36.

Davison, A. P., Brüderle, D., Eppler, J. M., Kremkow, J., Muller, E., Pecevski, D., Perrinet, L., and Yger, P. (2008). PyNN: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics*, 2(11).

Dayan, P. (1999). Unsupervised learning. *The MIT Encyclopedia of the Cognitive Sciences*.

Dayma, B., Patil, S., Cuenca, P., Saifullah, K., Abraham, T., Le Khac, P., Melas, L., and Ghosh, R. (2021). DALL-E Mini.

De Michell, G. and Gupta, R. (1997). Hardware/software co-design. *Proceedings of the IEEE*, 85(3):349–365.

Dellaferrera, G. and Kreiman, G. (2022). Error-driven input modulation: solving the credit assignment problem without a backward pass. In *International Conference on Machine Learning*, pages 4937–4955. PMLR.

Dellaferrera, G., Woźniak, S., Indiveri, G., Pantazi, A., and Eleftheriou, E. (2022). Introducing principles of synaptic integration in the optimization of deep neural networks. *Nature Communications*, 13(1):1–14.

Donati, E., Payvand, M., Risi, N., Krause, R., Burelo, K., Dalgaty, T., Vianello, E., and Indiveri, G. (2018). Processing EMG signals using reservoir computing on an event-based neuromorphic system. In *Biomedical Circuits and Systems Conference, (BioCAS)*, pages 1–4. IEEE.

Donati, E., Payvand, M., Risi, N., Krause, R., and Indiveri, G. (2019). Discrimination of EMG signals using a neuromorphic implementation of a spiking neural network. *Biomedical Circuits and Systems, IEEE Transactions on*, 13(5):795–803.

Douglas, R. J. and Martin, K. A. (2004). Neuronal circuits of the neocortex. *Annual Review of Neuroscience*, 27:419–451.

Douglas, R. J., Martin, K. A., and Whitteridge, D. (1989). A canonical microcircuit for neocortex. *Neural Computation*, 1:480–488.

Economist, T. (2020). The cost of training machines is becoming a problem. https://www.economist.com/technology-quarterly/2020/06/11/the-cost-of-training-machines-is-becoming-a-problem. Accessed: 2022-11-11.

Edler, D., Eriksson, A., and Rosvall, M. (2020). The mapequation software package. mapequation.org. Accessed: 2022-12-27.

Ehrlich, B. (2022). The brain in search of itself: Santiago ramón y cajal and the story of the neuron.

Eppler, J.-M., Helias, M., Muller, E., Diesmann, M., and Gewaltig, M.-O. (2008). Pynest: a convenient interface to the NEST simulator. *Fontiers in Neuroinformatics*, 12(2).

Erhan, D., Bengio, Y., Courville, A., Manzagol, P., Vincent, P., and Bengio, S. (2010). Why does unsupervised pre-training help deep learning? *The Journal of Machine Learning Research*, 11:625–660.

Fang, H., Shrestha, A., Zhao, Z., Wang, Y., and Qiu, Q. (2019). A general framework to map neural networks onto neuromorphic processor. In *20th International Symposium on Quality Electronic Design (ISQED)*, pages 20–25. IEEE.

Feldman, J. A. and Ballard, D. H. (1982). Connectionist models and their properties. *Cognitive science*, 6(3):205–254.

Furber, S. and Bogdan, P., editors (2020). *SpiNNaker: A Spiking Neural Network Architecture*. Boston-Delft: now publishers.

Furber, S., Galluppi, F., Temple, S., and Plana, L. (2014). The SpiNNaker project. *Proceedings of the IEEE*, 102(5):652–665.

Galluppi, F., Davies, S., Rast, A., Sharp, T., Plana, L. A., and Furber, S. (2012). A hierachical configuration system for a massively parallel neural hardware platform. In *Proceedings of the 9th conference on Computing Frontiers*, pages 183–192. ACM.

Gewaltig, M.-O. and Diesmann, M. (2007). Nest (neural simulation tool). *Scholarpedia*, 2(4):1430.

Gilson, M., Savin, C., and Zenke, F. (2015). Editorial: Emergent neural computation from the interaction of different forms of plasticity. *Frontiers in Computational Neuroscience*, 9.

Goodman, J. R. (1983). Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th annual international symposium on Computer architecture*, pages 124–131.

Gopalakrishnan, R., Chua, Y., and Kumar, A. J. S. (2019). Hardware-friendly neural network architecture for neuromorphic computing. *arXiv preprint arXiv:1906.08853*.

Gütig, R. (2016). Spiking neurons can discover predictive features by aggregate-label learning. *Science*, 351(6277).

Harris, J. A., Mihalas, S., Hirokawa, K. E., Whitesell, J. D., Choi, H., Bernard, A., Bohn, P., Caldejon, S., Casal, L., Cho, A., et al. (2019). Hierarchical organization of cortical and thalamic connectivity. *Nature*, 575(7781):195–202.

Hawkins, J., Ahmad, S., and Cui, Y. (2017). A theory of how columns in the neocortex enable learning the structure of the world. *Frontiers in Neural Circuits*, 11.

He, Y. and Evans, A. (2010). Graph theoretical modeling of brain connectivity. *Current opinion in neurology*, 23(4):341–350.

Hertz, J. (2022). Why SoCs need NoCs: Network on chip and the future of computing. https://www.allaboutcircuits.com/news/why-socs-need-nocs-network-on-chip-and-future-computing/, Accessed: 2023-01-17.

Horowitz, M. (2014). 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14.

Horvát, S., Gămănu t, R., Ercsey-Ravasz, M., Magrou, L., Gămănu t, B., Van Essen, D. C., Burkhalter, A., Knoblauch, K., Toroczkai, Z., and Kennedy, H. (2016). Spatial embedding and wiring cost constrain the functional layout of the cortical network of rodents and primates. *PLoS biology*, 14(7):e1002512.

Indiveri, G. and Sandamirskaya, Y. (2019). The importance of space and time for signal processing in neuromorphic agents. *IEEE Signal Processing Magazine*, 36(6):16–28.

Ito, T., Hearne, L., Mill, R., Cocuzza, C., and Cole, M. W. (2020). Discovering the computational relevance of brain network organization. *Trends in cognitive sciences*, 24(1):25–38.

Jamil, T. (1997). Ram versus cam. *IEEE Potentials*, 16(2):26–29.

Ji, Y., Zhang, Y., Chen, W., and Xie, Y. (2018). Bridging the gap between neural networks and neuromorphic hardware with a neural network compiler. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 448–460. ACM.

Ji, Y., Zhang, Y., Li, S., Chi, P., Jiang, C., Qu, P., Xie, Y., and Chen, W. (2016). Neutrams: Neural network transformation and co-design under neuromorphic hardware constraints. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 21. IEEE Press.

Kaiser, M. (2011). A tutorial in connectome analysis: topological and spatial features of brain networks. *Neuroimage*, 57(3):892–907.

Kato, S., Takeuchi, E., Ishiguro, Y., Ninomiya, Y., Takeda, K., and Hamada, T. (2015). An open approach to autonomous vehicles. *IEEE Micro*, 35(6):60–68.

Keckler, S. W., Dally, W. J., Khailany, B., Garland, M., and Glasco, D. (2011). Gpus and the future of parallel computing. *IEEE Micro*, 31(5):7–17.

Kerstjens, S. (2022). *The Tree-D Brain: The Connectome's Strategy for Developmental Self-Construction*. PhD thesis, ETH Zurich.

Koch, C. (2016). How the computer beat the go player. *Sci Am Mind*, 27:20–23.

Krause, R., van Bavel, J. J. A., Wu, C., Vos, M. A., Nogaret, A., and Indiveri, G. (2021). Robust neuromorphic coupled oscillators for adaptive pacemakers. *Scientific Reports*, 11(1).

Kreiser, R., Renner, A., Leite, V. R. C., Serhan, B., Bartolozzi, C., Glover, A., and Sandamirskaya, Y. (2020). An on-chip spiking neural network for estimation of the head pose of the icub robot. *Frontiers in Neuroscience*, 14.

Labbe, M. (2021). Energy consumption of ai poses environmental problems. https://www.techtarget.com/searchenterpriseai/feature/Energy-consumption-of-AI-poses-environmental-problems. Accessed: 2022-11-11.

Laughlin, S. B. and Sejnowski, T. J. (2003). Communication in neuronal networks. *Science*, 301(5641):1870–1874.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

Lemaréchal, C. (2012). Cauchy and the gradient method. *Doc Math Extra*, 251(254):10.

Li, X., Xu, F., Zhang, J., and Wang, S. (2013). A multilayer feed forward small-world neural network controller and its application on electrohydraulic actuation system. *Journal of Applied Mathematics*, 2013.

Lillicrap, T. P., Cownden, D., Tweed, D. B., and Akerman, C. J. (2016). Random synaptic feedback weights support error backpropagation for deep learning. *Nature Communications*, 7:13276.

Lin, C.-K., Wild, A., Chinya, G. N., Lin, T.-H., Davies, M., and Wang, H. (2018). Mapping spiking neural networks onto a manycore neuromorphic architecture. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI*, pages 78–89. ACM.

Liu, H. (2002). Routing table compaction in ternary cam. *IEEE Micro*, 22(1):58–64.

Liu, S.-C., Delbruck, T., Indiveri, G., Whatley, A., and Douglas, R. (2014). *Event-based neuromorphic systems*. Wiley.

Lynn, C. W. and Bassett, D. S. (2019). The physics of brain network structure, function and control. *Nature Reviews Physics*, 1(5):318–332.

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.

Mahowald, M. (1994). *An Analog VLSI System for Stereoscopic Vision*. Kluwer, Boston, MA.

McCulloch, W. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bull. Math. Biophys.*, 5:115–133.

Mead, C. (1990). Neuromorphic electronic systems. *Proceedings of the IEEE*, 78(10):1629–36.

Mead, C. (2020). How we created neuromorphic engineering. *Nature Electronics*, 3(7):434–435.

Mead, C. and Ismail, M. (1989). *Analog VLSI implementation of neural systems*, volume 80. Springer Science & Business Media.

Medsker, L. R. and Jain, L. (2001). Recurrent neural networks. *Design and Applications*, 5:64–67.

Merolla, P. A., Arthur, J. V., Alvarez-Icaza, R., Cassidy, A. S., Sawada, J., Akopyan, F., Jackson, B. L., Imam, N., Guo, C., Nakamura, Y., Brezzo, B., Vo, I., Esser, S. K., Appuswamy, R., Taba, B., Amir, A., Flickner, M. D., Risk, W. P., Manohar, R., and Modha, D. S. (2014). A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673.

Meunier, D., Lambiotte, R., and Bullmore, E. T. (2010). Modular and hierarchically modular organization of brain networks. *Frontiers in neuroscience*, 4:200.

Milde, M., Renner, A., Krause, R., Whatley, A. M., Solinas, S., Zendrikov, D., Risi, N., Rasetto, M., Burelo, K., and Leite, V. R. C. (2018). teili: A toolbox for building and testing neural algorithms and computational primitives using spiking neurons. `https://teili.readthedocs.io/en/latest/index.html`. Unreleased software, Institute of Neuroinformatics, University of Zurich and ETH Zurich.

Mill, R. D., Ito, T., and Cole, M. W. (2017). From connectome to cognition: The search for mechanism in human functional brain networks. *NeuroImage*, 160:124–139.

Mohamed, W. (2008). The edwin smith surgical papyrus: Neuroscience in ancient egypt. *IBRO History of Neuroscience*, 1.

Mohemmed, A., Schliebs, S., Matsuda, S., and Kasabov, N. (2013). Training spiking neural networks to associate spatio-temporal input–output spike patterns. *Neurocomputing*, 107:3–10.

Moradi, S., Qiao, N., Stefanini, F., and Indiveri, G. (2018). A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (DYNAPs). *IEEE Transactions on Biomedical Circuits and Systems*, 12(1):106–122.

Morita, T., Asada, M., and Naito, E. (2016). Contribution of neuroimaging studies to understanding development of human cognitive brain functions. *Frontiers in human neuroscience*, 10:464.

Mysore, N., Hota, G., Deiss, S. R., Pedroni, B. U., and Cauwenberghs, G. (2022). Hierarchical network connectivity and partitioning for reconfigurable large-scale neuromorphic systems. *Frontiers in Neuroscience*, 15.

Neckar, A., Fok, S., Benjamin, B. V., Stewart, T. C., Oza, N. N., Voelker, A. R., Eliasmith, C., Manohar, R., and Boahen, K. (2019). Braindrop: A mixed-signal neuromorphic architecture with a dynamical systems-based programming model. *Proceedings of the IEEE*, 107(1):144–164.

Neustadter, E., Mathiak, K., and Turetsky, B. (2016). Eeg and meg probes of schizophrenia pathophysiology. In *The neurobiology of Schizophrenia*, pages 213–236. Elsevier.

Nicola, W. and Clopath, C. (2017). Supervised learning in spiking neural networks with FORCE training. *Nature Communications*, 8(1):1–15.

Noda, H., Inoue, K., Kuroiwa, M., Igaue, F., Yamamoto, K., Mattausch, H. J., Koide, T., Amo, A., Hachisuka, A., Soeda, S., et al. (2005). A cost-efficient high-performance dynamic tcam with pipelined hierarchical searching and shift redundancy architecture. *IEEE Journal of Solid-State Circuits*, 40(1):245–253.

Numenta (2022). Ai is harming our planet: addressing ai's staggering energy cost. https://numenta.com/blog/2022/05/24/ai-is-harming-our-planet. Accessed: 2022-11-11.

Nunes, L. (2021). Totally wired: Where could we go with a map of the brain? *APS Observer*, 34.

OpenAI (2018). Ai and compute. https://openai.com/blog/ai-and-compute/. Accessed: 2022-11-11.

Pagiamtzis, K. and Sheikholeslami, A. (2006). Content-addressable memory (CAM) circuits and architectures: A tutorial and survey. *IEEE Journal of Solid-State Circuits*, 41(3):712–727.

Park, J., Yu, T., Joshi, S., Maier, C., and Cauwenberghs, G. (2016). Hierarchical address event routing for reconfigurable large-scale neuromorphic systems. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–15.

Perniola, L., Olivo, P., and Nowak, E. (2018). Experimental investigation of 4-kb RRAM arrays programming conditions suitable for TCAM. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26:2599–2607.

Ponulak, F. and Kasiński, A. (2010). Supervised learning in spiking neural networks with resume: sequence learning, classification, and spike shifting. *Neural Computation*, 22(2):467–510.

Pouget, A., Dayan, P., and Zemel, R. (2000). Information processing with population codes. *Nature Reviews Neuroscience*, 1(2):125–132.

Qiao, N., Mostafa, H., Corradi, F., Osswald, M., Stefanini, F., Sumislawska, D., and Indiveri, G. (2015). A reconfigurable on-line learning spiking neuromorphic processor comprising 256 neurons and 128k synapses. *Frontiers in neuroscience*, 9:141.

Rajendran, B., Cheek, R. W., Lastras, L. A., Franceschini, M. M., Breitwisch, M. J., Schrott, A. G., Li, J., Montoye, R. K., Chang, L., and Lam, C. (2011). Demonstration of cam and tcam using phase change devices. In *2011 3rd IEEE International Memory Workshop (IMW)*, pages 1–4. IEEE.

Ramesh, A., Dhariwal, P., Nichol, A., Chu, C., and Chen, M. (2022). Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125*.

Ranzato, M. and LeCun, Y. (2007). A sparse and locally shift invariant feature extractor applied to document images. In *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*, volume 2, pages 1213–1217. IEEE.

Rao, A., Plank, P., Wild, A., and Maass, W. (2022). A long short-term memory for AI applications in spike-based neuromorphic hardware. *Nature Machine Intelligence*, 4(5):467–479.

Reid, A. T., Headley, D. B., Mill, R. D., Sanchez-Romero, R., Uddin, L. Q., Marinazzo, D., Lurie, D. J., Valdés-Sosa, P. A., Hanson, S. J., Biswal, B. B., et al. (2019). Advancing functional connectivity research from association to causation. *Nature neuroscience*, 22(11):1751–1760.

Ríos, C., Youngblood, N., Cheng, Z., Gallo, M. L., Pernice, W. H. P., Wright, C. D., Sebastian, A., and Bhaskaran, H. (2019). In-memory computing on a photonic platform. *Science Advances*, 5(2):eaau5759.

Risi, N., Calabrese, E., and Indiveri, G. (2021). Instantaneous stereo depth estimation of real-world stimuli with a neuromorphic stereo-vision setup. In *International Symposium on Circuits and Systems, (ISCAS)*, pages 1–5. IEEE.

Roy, K., Jaiswal, A., and Panda, P. (2019). Towards spike-based machine intelligence with neuromorphic computing. *Nature*, 575(7784):607–617.

Rueckauer, B. and Delbruck, T. (2016). Evaluation of event-based algorithms for optical flow with ground-truth from inertial measurement sensor. *Frontiers in neuroscience*, 10(176).

Rueckauer, B., Lungu, I.-A., Hu, Y., Pfeiffer, M., and Liu, S.-C. (2017). Conversion of continuous-valued deep networks to efficient event-driven networks for image classification. *Frontiers in neuroscience*, 11:682.

Sahu, P. K. and Chattopadhyay, S. (2013). A survey on application mapping strategies for network-on-chip design. *Journal of systems architecture*, 59(1):60–76.

Sakurai, Y. (1996). Population coding by cell assemblies–what it really is in the brain. *Neuroscience research*, 26(1):1–16.

Sangiovanni-Vincentelli, A. and Martin, G. (2001). Platform-based design and software design methodology for embedded systems. *IEEE Design & Test of computers*, 18(6):23–33.

Scherer, D., Müller, A., and Behnke, S. (2010). Evaluation of pooling operations in convolutional architectures for object recognition. In *International conference on artificial neural networks*, pages 92–101. Springer.

Schmidhuber, J. (2014). Who invented backpropagation. *More [DL2]*.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117.

Schmidhuber, J. (2022). Annotated history of modern ai and deep learning. *arXiv preprint arXiv:2212.11279*.

Schultz, K. J. (1997). Content-addressable memory core cells a survey. *Integration*, 23(2):171–188.

Sebastian, A., Le Gallo, M., Khaddam-Aljameh, R., and Eleftheriou, E. (2020). Memory devices and applications for in-memory computing. *Nature Nanotechnology*, 15(7):529–544.

Sengupta, A., Ye, Y., Wang, R., Liu, C., and Roy, K. (2019). Going deeper in spiking neural networks: Vgg and residual architectures. *Frontiers in neuroscience*, 13:95.

Shannon, C. E. (2001). A mathematical theory of communication. *ACM SIGMOBILE mobile computing and communications review*, 5(1):3–55.

Song, S., Balaji, A., Das, A., Kandasamy, N., and Shackleford, J. (2020). Compiling spiking neural networks to neuromorphic hardware. In *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 38–50.

Sporns, O. and Betzel, R. F. (2016). Modular brain networks. *Annual review of psychology*, 67:613.

Stanojevic, A., Woźniak, S., Bellec, G., Cherubini, G., Pantazi, A., and Gerstner, W. (2022). An exact mapping from relu networks to spiking neural networks. *arXiv preprint arXiv:2212.12522*.

Stimberg, M., Brette, R., and Goodman, D. F. (2019). Brian 2, an intuitive and efficient neural simulator. *eLife*, 8:e47314.

Titirsha, T. and Das, A. (2020). Thermal-aware compilation of spiking neural networks to neuromorphic hardware. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 134–150. Springer.

Titirsha, T., Song, S., Balaji, A., and Das, A. (2021). On the role of system software in energy management of neuromorphic computing. In *Proceedings of the 18th ACM International Conference on Computing Frontiers*, pages 124–132.

Udvary, D., Harth, P., Macke, J. H., Hege, H.-C., de Kock, C. P., Sakmann, B., and Oberlaender, M. (2020). A theory for the emergence of neocortical network architecture. *BioRxiv*.

Urgese, G., Barchi, F., Macii, E., and Acquaviva, A. (2016). Optimizing network traffic for spiking neural network simulations on densely interconnected many-core neuromorphic platforms. *IEEE Transactions on Emerging Topics in Computing*, 6(3):317–329.

van den Heuvel, M. P. and Sporns, O. (2013). Network hubs in the human brain. *Trends in cognitive sciences*, 17(12):683–696.

Verma, N., Jia, H., Valavi, H., Tang, Y., Ozatay, M., Chen, L.-Y., Zhang, B., and Deaville, P. (2019). In-memory computing: Advances and prospects. *IEEE Solid-State Circuits Magazine*, 11(3):43–55.

Vincentelli, A. (2002). Platform-based design.

Vogelstein, R., Tenore, F., Philipp, R., Adlerstein, M., Goldberg, D., and Cauwenberghs (2003). Spike timing-dependent plasticity in the address domain. In *Advances in Neural Information Processing Systems (NIPS)*, volume 15, pages 1171–1178, Cambridge, MA, USA. MIT Press.

Wallace, G. K. (1992). The jpeg still picture compression standard. *IEEE transactions on consumer electronics*, 38(1):xviii–xxxiv.

Watts, D. J. and Strogatz, S. H. (1998). Collective dynamics of 'small-world' networks. *nature*, 393(6684):440–442.

Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560.

Witvliet, D., Mulcahy, B., Mitchell, J. K., Meirovitch, Y., Berger, D. R., Wu, Y., Liu, Y., Koh, W. X., Parvathala, R., Holmyard, D., et al. (2021). Connectomes across development reveal principles of brain maturation. *Nature*, 596(7871):257–261.

Zamarreño-Ramos, C., Linares-Barranco, A., Serrano-Gotarredona, T., and Linares-Barranco, B. (2012). Multicasting mesh aer: A scalable assembly approach for reconfigurable neuromorphic structured aer systems. application to convnets. *IEEE transactions on biomedical circuits and systems*, 7(1):82–102.

Zhang, J. and Tao, D. (2020). Empowering things with intelligence: a survey of the progress, challenges, and opportunities in artificial intelligence of things. *IEEE Internet of Things Journal*, 8(10):7789–7817.

Zhang, K. and Sejnowski, T. J. (2000). A universal scaling law between gray matter and white matter of cerebral cortex. *Proceedings of the National Academy of Sciences*, 97(10):5621–5626.

Zhang, W., Gao, B., Tang, J., Yao, P., Yu, S., Chang, M.-F., Yoo, H.-J., Qian, H., and Wu, H. (2020). Neuro-inspired computing chips. *Nature Electronics*, 3:371–382.

Zheng, P., Tang, W., and Zhang, J. (2010). A simple method for designing efficient small-world neural networks. *Neural Networks*, 23(2):155–159.

Zhu, X. and Klabjan, D. (2021). Continual neural network model retraining. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 1163–1171. IEEE.

Institute of Neuroinformatics
Prof. Dr. Giacomo Indiveri

Title of work:

Brain-Inspired Placement and Routing for Neuromorphic Processors

Thesis type and date:

Ph.D. Thesis, April 2023

Supervision:

Prof. Dr. Giacomo Indiveri

Student:

Name: Vanessa Rodrigues Coelho Leite
E-mail: vanessa@ini.uzh.ch