École doctorale des Sciences Pour l'Ingénieur de Lille - Nord De France

Improving Spiking Neural Networks Trained with Spike Timing Dependent Plasticity for Image Recognition

Doctoral Thesis
Computer Science

Pierre Falez

10 October 2019

Supervisors

Director
Pr. Pierre Boulet
Université de Lille

Co-director
Dr. Pierre Tirilly
Université de Lille

Co-supervisors
Dr. Ioan Marius Bilasco
Université de Lille

Dr. Philippe Devienne
CNRS

Jury

President
Pr. Nathalie H. Rolland
CNRS – Université de Lille

Referees
Dr. Timothée Masquelier
CERCO UMR 5549, CNRS – Université de Toulouse 3

Pr. Benoît Miramond
LEAT / CNRS UMR 7248 – Université Côte d'Azur

Examiner
Pr. Sander M. Bohte
Centrum Wiskunde and Informatica

CRIStAL
Centre de Recherche en Informatique, Signal et Automatique de Lille

iRCica

École doctorale des Sciences Pour l'Ingénieur de Lille - Nord De France

Amélioration des réseaux de neurones impulsionnels entraînés avec la STDP pour la reconnaissance d'images

Thèse en vue de l'obtention du titre de docteur en
Informatique et applications

Pierre Falez

10 Octobre 2019

Superviseurs

Directeur

Pr. Pierre Boulet
Université de Lille

Co-directeur

Dr. Pierre Tirilly
Université de Lille

Co-superviseurs

Dr. Ioan Marius Bilasco
Université de Lille

Dr. Philippe Devienne
CNRS

Jury

Présidente

Pr. Nathalie H. Rolland
CNRS – Université de Lille

Rapporteurs

Dr. Timothée Masquelier
CERCO UMR 5549, CNRS – Université de Toulouse 3

Pr. Benoît Miramond
LEAT / CNRS UMR 7248 – Université Côte d'Azur

Examinateur

Pr. Sander M. Bohte
Centrum Wiskunde and Informatica

CRIStAL
Centre de Recherche en Informatique,
Signal et Automatique de Lille

iRCica

Abstract

Computer vision is a strategic field, in consequence of its great number of potential applications which could have a high impact on society. This area has quickly improved over the last decades, especially thanks to the advances of artificial intelligence and more particularly thanks to the accession of deep learning. These methods allow machines to deal with complex tasks, to the point that they can surpass the human mind in some cases. Nevertheless, these methods present two main drawbacks in contrast with biological brains: they are extremely energy intensive and they need large labeled training sets. In regard to the energy problem, to be run these methods call for substantial infrastructures, which sometimes necessitate thousands of Watts. In comparison with the brain, it represents a huge gap, as this organ only requires around two dozens of Watts in total for all its activities. Using deep learning methods on portable devices is not viable at the moment. When it comes to data sets, the issue is entailed by the means of learning of algorithms, which are mainly supervised. This kind of algorithms has to know the information linked to the data to guide its learning process. Deep learning methods require a large number of labeled data, which entails laborious efforts to make such datasets. The development of unsupervised rules made this stage unnecessary.

Spiking neural networks (SNNs) are alternative models offering an answer to the energy consumption issue. One attribute of these models is that they can be implemented very efficiently on hardware, in order to build ultra low-power architectures. In return, these models impose certain limitations, such as the use of only local memory and computations. It prevents the use of traditional learning methods, for example the gradient back-propagation. Spike-timing-dependent plasticity (STDP) is a learning rule, observed in biology, which can be used in SNNs. This rule reinforces the synapses in which local correlations of spike timing are detected. It also weakens the other synapses. The fact that it is local and unsupervised makes it possible to abide by the constraints of neuromorphic architectures, which means it can be implemented efficiently, but it also provides a solution to the data set labeling issue. However, spiking neural networks trained with the STDP rule are affected by lower performances in comparison to those following a deep learning process. The literature about STDP still uses simple data (Modified-NIST (MNIST), ETH-80, NORB), but the behavior of this rule has seldom been used with more complex data, such as sets made of a large variety of real-world images.

The aim of this manuscript is to study the behavior of these spiking models, trained through the STDP rule, on image classification tasks. The main goal is to improve the performances of these models, while respecting as much as possible the constraints of neuromorphic architectures. The first contribution focuses on the software simulations of SNNs. Hardware implementation being a long and costly process, using simulation is a good alternative in order to study more quickly the behavior of different models. Nevertheless, developing software able to simulate efficiently spiking models is a tall order. Two simulators are worked on in this manuscript. Neural network scalable spiking simulator (N2S3) is the first one, it

was designed to be flexible, so it can simulate a large variety of models. Multiple approaches are tested on a motion detection task to shows this flexibility. The second simulator is convolutional spiking neural network simulator (CSNNS), that is optimized to simulate rapidly some models used in the development of this manuscript. A comparison is done with N2S3 to prove it efficiency.

Then, the contributions focus on the establishment of multi-layered spiking networks; networks made of several layers, such as those in deep learning methods, allow to process more complex data. One of the chapters revolves around the matter of frequency loss seen in several SNNs. This issue prevents the stacking of multiple spiking layers, since the fire frequency drops drastically throughout the layers. New mechanisms are offered to bypass this problem, while maintaining the performances of the network: a threshold adaptation rule, a neural coding, and a modified STDP rule. A study of these mechanisms is provided at the end of the chapter.

The center point then switches to a study of STDP behavior on more complex data, especially colored real-world images (CIFAR-10, CIFAR-100, STL-10). Several policies of on/off filtering are introduced, which enables SNNs to learn from RGB images. Then, multiple measurements are used, such as the coherence of filters or the sparsity of activations, to better understand the reasons for the performance gap between STDP and the more traditional methods. Sparse auto-encoders are used to draw these comparisons because these networks are one of the unsupervised learning methods with the wider range of utilization. Some avenues will be offered so that the performance gap of the two methods may be bridged. Preliminary results on the usage of whitening transformation show the potential of this pre-processing to increase performance on colored images (66.58% on CIFAR-10).

Lastly, the manuscript describes the making of multi-layered networks. To this end, a new threshold adaption mechanism is introduced, along with a multi-layer training protocol. A study of different mechanisms (STDP, inhibition, threshold adaptation) is provided at the end of the chapter. It is proven that such networks can improve the state-of-the-art for STDP on both MNIST (98.60%) and face/motorbikes (99.46%) datasets.

Résumé

La vision par ordinateur est un domaine stratégique, du fait du nombre potentiel d'applications avec un impact important sur la société. Ce secteur a rapidement progressé au cours de ces dernières années, notamment grâce aux avancées en intelligence artificielle et plus particulièrement l'avènement de l'apprentissage profond. Ces méthodes permettent de traiter des tâches complexes, au point de réussir à battre l'humain dans certains cas. Cependant, ces méthodes présentent deux défauts majeurs face au cerveau biologique : ils sont extrêmement énergivores et requièrent de gigantesques bases d'apprentissage étiquetées. Concernant le problème de l'énergie, ces méthodes ont besoin d'infrastructures conséquentes pour tourner, qui peuvent demander plusieurs milliers de watts. Cela représente un énorme fossé par rapport au cerveau, qui lui ne consomme qu'une vingtaine de watts pour la totalité de son fonctionnement. Embarquer ces méthodes artificielles dans des appareils portables n'est pas viable. Le problème des données est quant à lui causé par le mode d'apprentissage des algorithmes, qui sont majoritairement supervisés. Ce genre d'algorithme nécessite de connaître les informations associées aux données pour guider l'apprentissage. Étiqueter un grand nombre de données, comme le requière l'utilisation de méthode d'apprentissage profonds, représente un coût important. Le développement de règles non-supervisées permet de se passer de la nécessité d'étiqueter les données.

Les réseaux de neurones à impulsions sont des modèles alternatifs qui permettent de répondre à la problématique de la consommation énergétique. Ces modèles ont la propriété de pouvoir être implémentés de manière très efficace sur du matériel, afin de créer des architectures très basse consommation. En contrepartie, ces modèles imposent certaines contraintes, comme l'utilisation uniquement de mémoire et de calcul locaux. Cette limitation empêche l'utilisation de méthodes d'apprentissage traditionnelles, telles que la rétro-propagation du gradient. La STDP est une règle d'apprentissage, observé dans la biologie, qui peut être utilisée dans les réseaux de neurones à impulsions. Cette règle renforce les synapses où des corrélations locales entre les temps d'impulsions sont détectées, et affaiblit les autres synapses. La nature locale et non-supervisée permet à la fois de respecter les contraintes des architectures neuromorphiques, et donc d'être implémentable de manière efficace, mais permet également de répondre aux problématiques d'étiquetage des base d'apprentissages. Cependant, les réseaux de neurones à impulsions entraînés grâce à la STDP souffrent pour le moment de performances inférieures aux méthodes d'apprentissage profond. La littérature entourant la STDP utilise très majoritairement des données simples (MNIST, ETH-80, NORB), mais le comportement de cette règle n'a été que très peu étudié sur des données plus complexes, tel que sur des bases avec une variété d'images importante.

L'objectif de ce manuscrit est d'étudier le comportement des modèles impulsionnels, entraîné via la STDP, sur des tâches de classification d'images. Le but principal est d'améliorer les performances de ces modèles, tout en respectant un maximum les contraintes imposées par les architectures neuromorphiques. Une première partie des contributions proposées dans ce manuscrit s'intéresse à la simulation logicielle des réseaux de neurones impulsionnels. L'implémentation

matérielle étant un processus long et coûteux, l'utilisation de simulation est une bonne alternative pour étudier plus rapidement le comportement des différents modèles. Cependant, développer des logiciels capables de simuler efficacement les modèles impulsionnels représente un défi de taille. Deux simulateurs sont proposés dans ce manuscrit: le premier, N2S3, est conçu pour être flexible, et donc permet de simuler une très grande variété de modèles. Afin de démontrer la flexibilité de ce simulateur, plusieurs approches sont utilisées sur une tâche de détection de mouvement. Le second simulateur, CSNNS, est quand à lui optimisé pour simuler rapidement certains modèles utilisés dans la suite de ce manuscrit. Une comparaison entre N2S3 et CSNNS est effectuée afin de prouver son efficacité.

La suite des contributions s'intéresse à la mise en place de réseaux impulsionnels multi-couches. Les réseaux composées d'un empilement de couches, tel que les méthodes d'apprentissage profond, permettent de traiter des données beaucoup plus complexes. Un des chapitres s'articule autour de la problématique de perte de fréquence observée dans les réseaux de neurones à impulsions. Ce problème empêche l'empilement de plusieurs couches de neurones impulsionnels, car la fréquence de décharge des neurones chute de manière drastique. Sans activité dans une couche, aucun apprentissage ne peut se faire puisque la STDP ne s'effectuera pas. De nouveaux mécanismes sont introduits pour répondre à ce problème, tout en conservant les performances de reconnaissance : une règle d'adaptation du seuil, un codage d'entrée ainsi qu'une règle STDP modifiée. L'étude de ces mécanismes est faite à la fin du chapitre.

Une autre partie des contributions se concentre sur l'étude du comportement de la STDP sur des jeux de données plus complexes, tels que les images naturelles en couleurs (CIFAR-10, CIFAR-100, STL-10). Dans cette partie sont proposées plusieurs politiques de filtrage on/off qui permettent aux réseaux de neurones impulsionnels d'apprendre sur des images RBV. Plusieurs mesures sont utilisées, telle que la cohérence des filtres ou la dispersion des activations, afin de mieux comprendre les raisons de l'écart de performances entre la STDP et les méthodes plus traditionnelles. Les auto-encodeurs épars sont utilisés dans cette comparaison car ils sont une des méthodes non-supervisées les plus utilisées. Plusieurs pistes sont ainsi proposées afin de réduire le fossé entre les performances des deux méthodes. La fin du chapitre montre des résultats préliminaires qui suggèrent que l'utilisation du whitening permet d'améliorer nettement les performances de la STDP sur les images couleurs (66.58% sur CIFAR-10).

Finalement, la réalisation de réseaux multi-couches est décrite dans la dernière partie des contributions. Pour ce faire, un nouveau mécanisme d'adaptation des seuils est introduit ainsi qu'un protocole permettant l'apprentissage multi-couches. L'étude des différents mécanismes (STDP, inhibition) est fournie à la fin du chapitre. Il est notamment démontré que de tels réseaux parviennent à améliorer l'état de l'art autour de la STDP sur les bases d'apprentissage MNIST (98.60%) et face/motorbikes (99.46%).

Acknowledgements

The manuscript you are about to read was the work of three years: three years during which I was helped and supported by many people. First I would like to thank the University of Lille for its financial support without which I could not have done my PhD. For the office materials and business trips, which allowed me to enlarge my vision of scientific fields, I have to thank CRIStAL. Obviously, the IRCICA institute cannot be forgotten. For three years, I worked in one of their office, on one of their computer, using their computation resources, with members of their teams being myself one. I am also so grateful for the plethora of business travels through which I entered the scientific world and met researchers. Those discoveries and experiences lead to the broadening of my horizon and to the development of my scientific thinking.

There is no need to say that, in addition to the materials support, the human guidance I received was a key element in the completion if this manuscript. The first man to thank is my thesis advisor, Pr. Pierre Boulet, together with my three co-supervisors, Dr. Pierre Tirilly, Dr. Ioan Marius Bilasco and Dr. Phillipe Devienne. They trained me and helped me to become the man I am today. They brought me, at once, advice and solutions on the issues I encountered during the thesis. They also taught me how to communicate and explain my works to others, whether it was oral or written. They read again and again this manuscript and the other papers I published, showing an infinite number of times their involvement and seriousness. They were models to follow.

Next is the thesis committee. First, my referees, Dr. Timothée Masquelier and Pr. Benoît Miramond, took the time to read carefully this manuscript, providing me with precise and helpful guidance. Their help definitely improved the manuscript and, with it, the thesis presentation. I am grateful for their numerous questions and professionalism. Then, I want to thank the jury president, Pr. Nathalie H. Rolland for presenting and presiding the presentation of my thesis. In addition, my examiner, Pr. Sander M. Bohte, took the time to read minutely my thesis and came up with refreshing and interesting questions and suggestions.

A special thank you for my girlfriend, Audrey Magnier, who supported me through the all thesis. She also took the time to proofread the manuscript and to improve my presentations. For her expertise on English and languages, I can not thank her enough. She succeeded in cheering me up in difficult times. I also want to thank my family and especially my parents, Jean-Yves and Laurence Falez, for supporting me in the path of I.T and for always believing in me. And finally, a big thank you to all my colleagues and friends for their kindness and conviviality. It is impossible to stay down around them. You all helped me to look up when I faced adversity. In particular, thank you to the colleagues I shared my office with, Houssam Eddine-Zahaf, Marco Pagani, Fabien Bouquillon and Frédéric Fort. Each day I worked with you was a pleasure and a new experience. I learned so much in the talks around coffees. I also have to mention my closest friends, among whom Benjamin Danglot, Phillipe Antoine, Nicolas Cachera, Aurélie Desmet, Alexandre Verkyndt, Quentin Baert, and Camille Marquette. I cannot forget the other PhD students, from my doctoral school, from the PhD candidate association of Lille (ADSL), and from the GDR Biocomp, with whom I've had so many good times.

To all these people and those I did not mention but that I do not forget, sincerely, thank you.

– Pierre Falez

Contents

Chapter 1

Introduction, context, and motivations

Due to its large number of potential applications, computer vision is a strategic area, whether it is to guide an autonomous vehicle, to inspect production lines in factories, to monitor unusual events, to diagnose diseases from medical imaging... All these applications can be easily mastered by humans, thanks to the brain which has evolved in order to deal with the critical task that is to improve the survival of the species [1]. However, creating artificial methods capable of competing with the human brain is very challenging. Much progress has been made in this area since the apparition of early artificial approaches such as the perceptron in 1957. Nowadays, advanced artificial methods, such as artificial neural networks (ANNs) and, especially, deep learning [2], are able to compete with humans on multiple tasks. To achieve this feat, such methods are fed with millions of sample images, in order for them to learn how to generalize to new data.

As an example, to make such a model recognize whether an image contains a dog or a cat, it is necessary to train it with many examples. Thus, it is necessary to retrieve thousands of images of cats and dogs under different poses, of different races, in different contexts...Each image provided to the model necessitates a label (i.e. dog or cat) in order to guide the algorithm during the learning process. When the model predicts a wrong label, it has to adjust its parameters in order to give a better prediction the next time it is presented with this image. By iterating this process thousands or millions of times, the model should find parameters that allow it to succeed in this task. However, these methods remain at a disadvantage on several points compared to the brain. A first drawback is that artificial methods tend to be task-specific: ANNs are able to recognize only the objects on which they have been trained. These models have difficulties to evolve. To add new classes to a trained model, it is generally necessary to start learning again from the beginning and sometimes to change the network architecture. In opposition, the brain adapts itself in order to learn new objects. Moreover, when it learns to recognize a new object, it can do so with very few examples (i.e. sometimes a single example is enough, this ability is called one-shot learning [3], [4]), and without restarting the learning from the beginning. A second drawback is the supervision required by these methods. Each sample provided to the network needs to be labeled. This involves a lot of efforts in order to label large databases as required by deep learning [5]. Ensuring a low rate of mislabeled data is also an endeavoring job. Labeling can be tedious. Sometimes it requires the presence of experts for certain complex tasks, which is costly. The brain works differently: it can learn by observing and interacting with the environment, thanks to its ability to use unsupervised and reinforcement learning [6]. A last drawback is the energy inefficiency of the artificial methods [5], [7], [8]. Currently, models allowing to solve the most complex tasks require huge infrastructures to run. Compared to the brain, that consumes about 20 W to work [9], [10], artificial methods can require hundreds or thousands of Watts. The massive use of this kind of models would require a huge amount of energy. Lots of companies have offered optimized hardware to run artificial models more efficiency [11], but those can not be embeded directly into mobile devices, such as smartphones or Internet of things (IoT) sensors, due to their high energy consumption.

This energy gap between the artificial methods and the brain is especially due to their operating models. The brain uses massively desynchronized units with only local computation and memory. Artificial methods use the von Neumann architecture, which works in a synchronized mode of operation, with the memories separated from the computation units. Moreover, the performances of these models are limited by the von Neumann bottleneck: the performance of the system is limited by the amount of data that can be exchanged on the bus between the computation unit and the memory.

Von Neumann architectures are also exposed to the end of Moore's law [12]. In order to continue to improve the computational power of machines, new alternatives must be developed. Beside quantum or optical computing, neuromorphic architectures are a promising alternative to von Neumann architectures [13]. Neuromorphic computing allows dealing with cognitive tasks, while remaining much more efficient than von Neumann architectures [14]. By using a mode of operation inspired from the brain, these architectures aim to improve artificial method by taking the advantages observed in biology. Studies show that neuromorphic platforms can consume up to 300,000 times less energy than traditional von Neumann architectures [15]. Neuromorphic architectures are cited as a breakthrough technologies by the MIT[1] and as a highly promising technology by the U.S. government [16].

1.1 Neuromorphic Hardware

The spiking neural network (SNN) is a model that is part of neuromorphic computing. Unlike ANNs, that perform computation on numerical representations, this model uses spikes to encode and process information. SNNs are related to hardware implementation, since one of their main advantages is to allow the development of energy efficient hardware. Three different approaches are possible: digital hardware, analog hardware or mixed digital/analog hardware. In order to provide efficient implementations, these architectures must be massively parallel and weakly synchronized [17], and must have an optimized routing system between all the units of the system to avoid the von Neumann bottleneck. However, such architectures enforce some constraints, such as the locality of the operations, which can prevent the usage of some mechanisms.

The main advantage of digital approaches is the simplicity of their design over analog architectures [18]. Unlike analog approaches, simulated models can be noiseless. Moreover, digital architectures have a good ability to scale up to large networks. Such architectures use classical boolean logic gates, and most of the time, one or multiple clocks to synchronize computations. TrueNorth [15], SpiNNaker [19], or Loihi [20] are projects that already provide digital neuromorphic architectures. In order to provide efficient simulations, these architectures use multiple cores connected to each other, each of them able to simulate multiple neurons and synapses. Such hardware uses an optimized routing system that allows low-cost communications within the network [21].

Field-programmable gate array (FPGA) are another alternative to implement digital neuromorphic architectures [22]–[24]. They allow shorter design and fabri-

[1] https://www.technologyreview.com/s/526506/neuromorphic-chips/

cation durations by providing reconfigurable architectures which can be optimized for different tasks. FPGA can be easily interfaced with host computers or with other FPGA [25], [26].

Analog hardware has the best potential to produce ultra-low-power chips, by directly using the physics of silicon to reproduce the behavior of SNNs [27]. Another advantage of analog design is its compactness: it requires a smaller area in the integrated circuit [18]. One disadvantage of the analog approach compared to the digital one is its lower signal-to-noise ratio and the variability of components [18]. However, some work suggests that SNNs are tolerant to these variations [28], so analog implementation can be suited to neuromorphic computing. Some studies even report that the noise which can be brought by the hardware components improves learning in SNN [29]. Another issue with the analog approach is that it is less flexible: for example, time constants must be fixed in the design, and cannot be modified during the simulation. Some work uses complementary metal-oxide semiconductor (CMOS) technology, which has the advantage to have a stable fabrication process. These circuits can be integrated into very large-scale integration (VLSI) in order to build large and dense chips. However, many authors use new microelectronics technologies, with greater potentials (e.g. energy efficient, compactness...) than CMOS, notably memristive devices, which have a variable internal resistance based on the previous current that flowed through them [30]. This behavior is interesting for neuromorphic computing, since synapses can also be considered as a memristive system [31].

Leon Chua predicted in 1971 the memristor [32] as the fourth elemental passive component. The first practical realization was made by HP labs in 2008 [33] with a thin titanium dioxide film (TiO_2). It should be noted that the usage of memristive devices is not limited to synapses, but can also be employed for neuron modeling [34], [35]. Multiple technologies are in competition to implement memristive devices: resistive RAM (ReRAM) [36], phase change memory (PCM) [37], spin-torque transfer RAM (STT-RAM) [38], nanoparticle organic memory field-effect transistor (NOMFET) [39]...

Field-programmable analog array (FPPA), the analog equivalent of FPGA, is another interesting technology to implement analog neuromorphic architectures [40]–[42]. They bring advantages similar to FPGA, such as reduced design duration and cost. However, only small networks can be currently implemented on FPPA, a few dozen neurons at most, due to the reduced number of available functional blocks on commercialized chips.

Finally, some projects try to combine digital and analog implementations, in order to bypass the disadvantages of the two approaches. However, it is necessary to have compatible technologies in order to use both analog and digital components within the same architecture. BrainScaleS [43], Neurogrid [44], and ROLLS [45] are projects of mixed analog/digital architectures. These projects use analog circuits to simulate neurons, and thus, benefit from their energy efficiency, and digital communication systems to facilitate the scaling properties of such architectures.

1.2 Neuromorphic Constraints

Neuromorphic architectures have the potential to significantly reduce the energy consumption of learning systems. However, this kind of architectures achieves such performance by enforcing some constraints. These constraints must be taken into account because they enforce the mechanisms that can be used with SNNs. The major one, which is present in the vast majority of neuromorphic hardware, is the locality of computation and memory. This locality varies in the different architectures: in some digital hardware, locality is restricted to a core, but in others, notably in analog architectures, the locality is limited to a single neuron. This constraint limits the algorithms that can be used, and thus, finding effective local learning rules is crucial in order to compete with traditional artificial methods. Spike-timing-dependent plasticity (STDP) is a popular rule in the SNN community that meets this requirement [10].

Another constraint is the type of communication allowed in the network. Some architectures allow transmitting small packets, which can contain a few bytes of information, while others support only spikes as a means of communication. In addition, the connectivity can be restricted. Some architectures use fixed topologies, notably when using analog connections. Reconfigurable topology architectures can also impose some constraints, such as the maximum number of incoming or outgoing connections per neuron, or the possibility of connecting neurons that are not located in the same part of the system.

Some architectures tend to be flexible, and allow simulating a significant amount of neuron and synapse models, but others are limited to specific types, with more or less flexibility on the setting of model parameters. The different contributions reported in this manuscript are motivated by the advantages provided by the neuromorphic architectures. Thus, a particular attention is given to these various constraints mentioned above, in order to make the contributions suitable to hardware implementations.

1.3 Motivations

Computer vision applications are of great interest in many sectors, and are likely to be increasingly used in the coming years. With the parallel democratization of the IoT, such applications should be embedded in mobile devices. Currently, lots of applications send requests to distant deep models because they are not usable on such devices. If the inference of such models could be done locally, it would reduce the energy consumption, but also improve the data privacy. Moreover, implementing local training is another important challenge, since some applications adapt themselves to the user habits. Bio-inspired architectures, such as neuromorphic computing, are trying to meet this challenge. In return, such hardware enforces some limitations, which prevent the usage of traditional learning methods. However, learning rules which respect these constraints, like STDP, have not yet succeeded to compete this advanced artificial methods, like deep learning. The motivation of this manuscript is to study the behavior of SNNs trained with STDP on image classification tasks, in order to offers new mechanisms that improve their performance, while trying to respect the constraints of the neuromorphic architectures. Notably, it is recognized that using deep hierarchical representations improves the expressiveness of models [46], and yields state-of-the-art performance on many tasks [47], [48]. Succeeding in training a multilayer SNN with STDP is an important objective which can reduce the gap between ANNs and SNNs. A second motivation is the processing of complex data. Currently, most of the STDP literature uses simple datasets, such as Modified-NIST (MNIST), and only little work is interested in the use of more complex datasets. In addition, this manuscript also addresses the problem of the software simulation of SNNs: since larger networks should be used to process more complex data, it is important to designing simulators that speed up experimentation while remaining flexible.

1.4 Outline

This manuscript gives in Chapter 2 an overview the different domains necessary for understanding the contributions. Notably, this chapter offers an introduction to object recognition (Section 2.1) and spiking neural networks (Section 2.2), but also the literature of image classification with SNNs (Section 2.3) and software simulation of SNN (Section 2.4). Then the contributions of this manuscript are detailed. Chapter 3 focuses on the software simulation of SNNs, and presents the tools developed in order to simulate the models proposed in the manuscript. The first simulator is the neural network scalable spiking simulator (N2S3) (Section 3.1). A case study is detailed to show the flexibility brought by this simulator. The second tool is the convolutional spiking neural network simulator (CSNNS) (Section 3.2), which allows to efficiently simulate specific models of SNNs. Then, Chapter 4 points out the problem of frequency loss observed with SNNs. This issue is critical to multilayer SNNs, since the network is not able to learn efficiently without a sufficient activity. Mechanisms are offered in the remaining of the chapter in order to maintain spike frequency across the layers, but also the recognition rate of the network. Chapter 5 introduces mechanisms that allow SNNs to learn patterns on colored images, and evaluates the performance of STDP on complex datasets. A second purpose of this chapter is to compare the quality of the features learned by STDP to features learned by auto-encoders (AEs), and to discuss their differences in order to open up new perspectives. Finally, Chapter 6 extends the mechanisms introduced in the previous chapter and provides a protocol that allows to train multilayer SNNs with STDP rules. Then, this chapter shows that multilayer SNNs trained with STDP lead to state-of-the-art results on both MNIST and the faces/motorbikes dataset.

Chapter 2

Background

The subject of this manuscript covers several disciplines, such as computer vision, software simulation, or neurosciences, thus an introduction to the different related fields is necessary. We first discuss the application, with the introduction of the object recognition task in order to better understand the challenges of our work, in Section 2.1. We give an overview of the different methods and datasets used in this field, and of the state-of-the-art in order to be able to position the different contributions. Afterward, in Section 2.2, we provide a review of the spiking neural network models, and of the different mechanisms used in the literature. Then, in Section 2.3, we focus on the learning methods employed to train spiking neural networks on images classification tasks, and we list the best results currently achieved on the different datasets. Finally, in Section 2.4 we outline the software simulator available to run spiking neural network (SNN), and the underlying challenges.

2.1 Object Recognition

Object recognition is a challenging computer vision task studied for decades. It is an attractive research field due to the numerous applications: industrial manufacturing requirements (e.g. quality inspection, object sorting and counting), medical imaging (e.g. diagnostic from radiography), security (e.g. unusual event detection from cameras). Object recognition regroups several tasks, such as localization (i.e. find position of object instances), segmentation (i.e. associate a label to a group of pixels), or classification. This manuscript focuses on the image classification task, because it is one of the most studied, due to the large number of possible applications. This task consists in associating a pre-defined class (e.g. dog, car, flower) to an image, that best describes the content of the image. In some classification tasks, multiple labels can be associated with a single image if multiple objects are present in it. This manuscript is focused only on associating a single class with an image, since this task is already very challenging for the studied models. Object recognition tasks can seem easy to humans, because the brain is highly efficient in processing visual information, but remains challenging for artificial methods. One reason is that intra-class variations can be huge [1]: two objects belonging to the same class can have a variety of model instances, but also position, scale, lighting, background, and pose variations (see Figure 2.1).

Formally speaking, an image $\mathbf{X}$ is a matrix of pixels. Each pixel is generally represented by one value, as in grayscale format (see Figure 2.2), or by three values, as in the red green blue (RGB) format. Thus each image $\mathbf{X}$ is a 3D array:

$$\mathbf{X} \in [0,1]^{x_{\text{width}} \times x_{\text{height}} \times x_{\text{depth}}} \tag{2.1}$$

with x_{width} and x_{height} the width and height of the image, and x_{depth} the number of components per pixel.

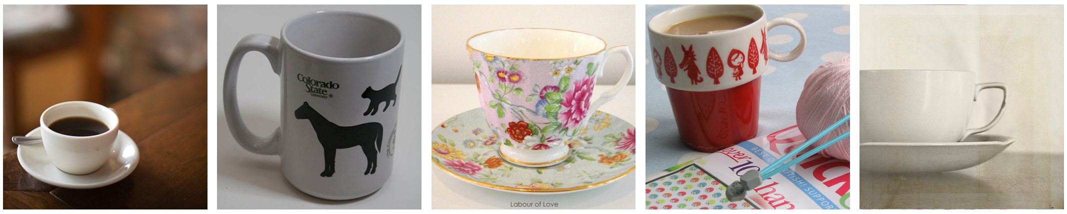
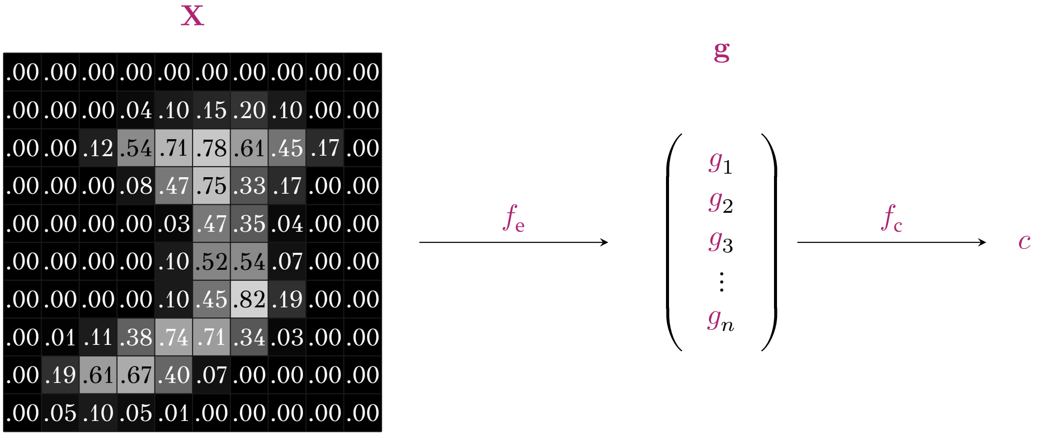


Figure 2.1: Examples of variations between instances of the same object class (cup). There are positions, scales, poses, lighting, and background variations, but also a variety of model instances (e.g. tea cup, coffee cup, mug...).



.00	.00	.00	.00	.00	.00	.00	.00	.00	.00
.00	.00	.00	.04	.10	.15	.20	.10	.00	.00
.00	.00	.12	.54	.71	.78	.61	.45	.17	.00
.00	.00	.00	.08	.47	.75	.33	.17	.00	.00
.00	.00	.00	.00	.03	.47	.35	.04	.00	.00
.00	.00	.00	.00	.10	.52	.54	.07	.00	.00
.00	.00	.00	.00	.10	.45	.82	.19	.00	.00
.00	.01	.11	.38	.74	.71	.34	.03	.00	.00
.00	.19	.61	.67	.40	.07	.00	.00	.00	.00
.00	.05	.10	.05	.01	.00	.00	.00	.00	.00

Figure 2.2: Image classification process. Images are arrays of pixels (e.g. here in grayscale format, with the highest values representing the brightest pixels). The feature extractor f_e is responsible for transforming this representation into a vector $\mathbf{g}$, containing the different features. Finally, the classifier f_c predicts a class c, among all possible classes $\mathcal{C}$, from $\mathbf{g}$.

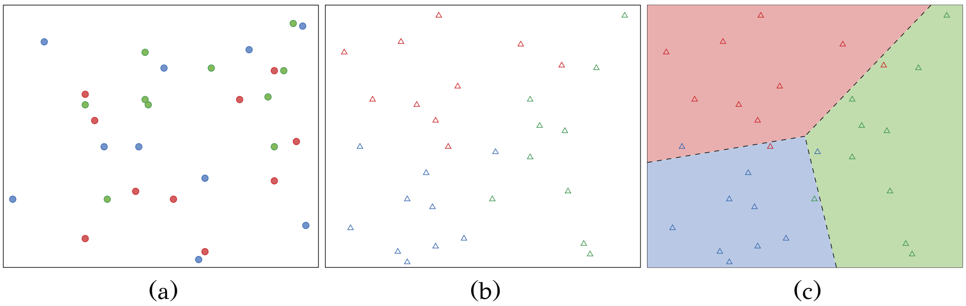


Figure 2.3: Image classification process. (a) In the pixel space, classes can not be easily separated. (b) However, in the feature space the inter-class variations can become larger than intra-class variations, (c) which allows finding boundaries that allows a correct separation thanks to a classifier.

The classification task consists in associating a label c, from a set of n possible classes $\mathcal{C} = \{c_0, c_1, \dots, c_n\}$ to an image $\mathbf{X}$:

$$\begin{aligned} f_{\text{ec}} : \mathbb{R}^{x_{\text{width}} \times x_{\text{height}} \times x_{\text{depth}}} &\to \mathcal{C} \\ \mathbf{X} &\mapsto c \end{aligned} \tag{2.2}$$

Classification requires to find boundaries between classes. However, finding boundaries that separate the different classes correctly in the pixels space is non-trivial due to the intra-class variations that can be as large as the inter-class variations. So, before doing the classification, it is most of the time necessary to map the image into a feature space that allows a better separation of the classes thanks to a feature extractor f_{e}:

$$\begin{aligned} f_{\text{e}} : \mathbb{R}^{x_{\text{width}} \times x_{\text{height}} \times x_{\text{depth}}} &\to \mathbb{R}^{n_{\text{features}}} \\ \mathbf{X} &\mapsto \mathbf{g} \end{aligned} \tag{2.3}$$

with $\mathbf{g}$ the feature vector of dimension n_{features}. It defines a dictionary of features of size n_{features}.

This step consists in disentangling the data, typically to make the data linearly separable. This transformation can be done by extracting several features from the image [49]. Pixels are the lowest level of features, but by applying a sequence of operations, it is possible to get a higher level of features (i.e. more abstract features), which may be invariant to some properties (orientation, scale, illumination...), and so, more informative for the classification (see Figure 2.3).

Then, a classifier f_{c} can be used on the feature vector $\mathbf{g}$ to predict a class c:

$$\begin{aligned} f_{\text{c}} : \mathbb{R}^{n_{\text{features}}} &\to \mathcal{C} \\ \mathbf{g} &\mapsto c \end{aligned} \tag{2.4}$$

Thus, the image classification can be expressed as $f_{\text{ec}} = f_{\text{c}} \circ f_{\text{e}}$.

In the following, Section 2.1.1 enumerates the traditional features extracted from images and Section 2.1.2 lists the main classification methods. Section 2.1.3 focuses on multi-layer neural networks, which have the ability to learn both the features to extract and the boundaries between classes. Finally, section 2.1.4 presents different datasets frequently used in image classification.

2.1.1 Feature Extraction

Finding features that improve the invariance of the model is crucial to create an effective image classification system [49]. Features can be separated into two categories. On the one hand, processes that use all the pixels of the image generate

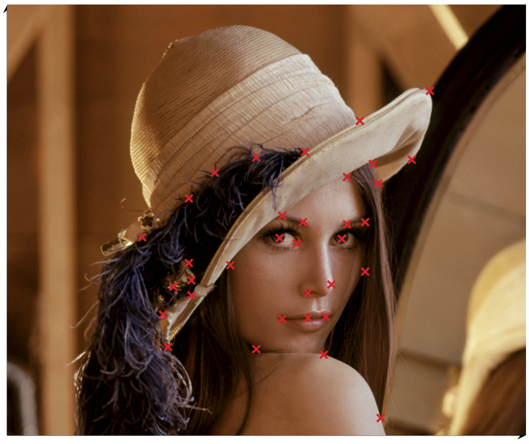


Figure 2.4: Examples of interest points in an image.

global features, such as color histograms, texture descriptors or, spectral analysis. Such features often have the advantage of being fast to compute, and being compact in memory. On the other hand, local features describe specific parts of the image, by characterizing only pixels in some region of interest or around densely sampled regions [50], [51]. Some methods, such as corner or blob detector, allow extracting a set of interest points (i.e. sometimes called key points or salient points) in the images that have the potential to be informative (Figure 2.4). Some detectors work at specific scales (e.g. Harris detector, FAST detector...), while other methods allow multi-scale detection (i.e. Laplacian-of-Gaussian, Gabor-Wavelet detector...). Then, a set of descriptors can be computed for each interest point, such as the local gradient. Local features have the advantage of being able to provide a more detailed description of the image. Among the most common methods used to extract local features, the scale-invariant feature transform (SIFT) uses the difference of Gaussians (DoG) operator as an interest point detector. It then extracts the dominant orientations around each point. Speeded-up robust features descriptor (SURF) is another method; it uses an approximation of the determinant of the Hessian blob detector to detect interest points and use Haar wavelet responses around it as a descriptor.

Since the rise of deep learning, features tend to be learned by algorithms rather than designed by human effort. Features learned from data have shown their superiority on a number of tasks, such as image classification [52], image segmentation [53], and action recognition [54]. Moreover, hierarchical models tend to combine some low-level features to generate more abstract features. Each level of features gives the ability to increase the invariance of the system.

2.1.2 Classification Methods

The key role of the classifier is to generalize the representation of the different classes to unseen images. To this end, classifiers need to find boundaries in the feature space that best separate classes. Thus, effective features are required to improve this separation. The classification process requires a training step. A training set, composed of the images $\mathcal{X}_{\text{train}}$ and the associated labels $\mathcal{Y}_{\text{train}}$, is used to create a model. Once the training is complete, the classifier should be able to predict the class of unseen samples. In order to test the performance of the classifier, a test step is performed: a test set, composed of both images $\mathcal{X}_{\text{test}}$ and labels $\mathcal{Y}_{\text{test}}$, which are not used during training, is used to measure the generalization capacity of the model (see Figure 2.5). One performance measure is the classification rate, which gives the ratio of good predictions over the total number of samples in the test set:

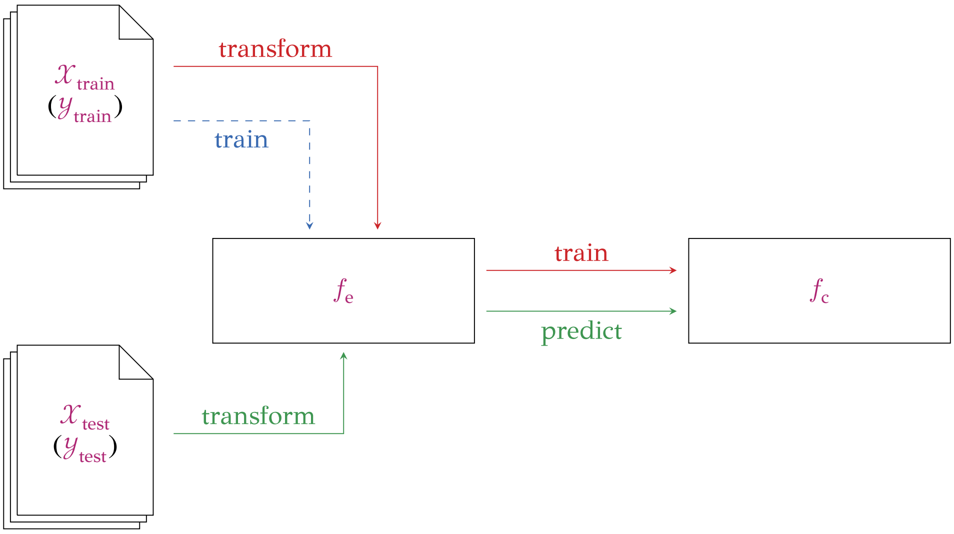

$\mathcal{X}_{\text{train}}$
$(\mathcal{Y}_{\text{train}})$
transform
train
f_e
train
predict
f_c
$\mathcal{X}_{\text{test}}$
$(\mathcal{Y}_{\text{test}})$
transform

Figure 2.5: The first step in an image classification task consists in training the model. In models which learn the parameters of the feature extractor from the data, the feature extractor is trained from the training set (blue pathway). Then, the classifier is trained from the features extracted from the training set (green pathway). Finally, the performance of the model can be evaluated by using the classifier to predict the class of each sample in the test set (red pathway).

$$\text{rr} = \frac{\sum\limits_{\mathbf{X} \in \mathcal{X}_{\text{test}}, c \in \mathcal{Y}_{\text{test}}} [f(\mathbf{X}) = c]}{|\mathcal{X}_{\text{test}}|} \tag{2.5}$$

Under-fitting and over-fitting are two phenomena that can explain the poor performance of a classifier. Under-fitting means that the classifier is not able to generalize to new data because the model is not complex enough to represent the data well (e.g. when using a linear classifier on a non-linearly separable problem). In this case, both the accuracies on the training set and on the testing set are low. In opposition, over-fitting appears when the classifier is not able to generalize well because it learns a representation too close to the training samples. In this case, the classifier has a very good accuracy on the training set, but a poor accuracy on the test set.

There are a lot of classifiers (e.g. the perceptron, decision trees, random forests...), but not all of them are able to perform well in the image classification context. One of the most used classifiers for this task, before the advent of the neural networks, was the support vector machine (SVM). SVM uses a set of hyperplanes which maximizes the margin between the classes. In its basic form, SVM is a linear classifier. A transformation into a high-dimensional space is possible according to a defined kernel to make non-linear classifications. In addition to the used kernel, SVM depends on a cost parameter svm_c, which allows tuning the optimization criterion between over-fitting and under-fitting. However, predictions made with this method are not so easily understandable.

However, the improvement brought by artificial neural networks (ANNs) makes the use of previous feature extraction and classification methods less relevant in many cases, particularly in object recognition.

2.1.3 Artificial Neural Network

ANNs are a family of models that are able to learn directly the features but also the class boundaries from the data. Indeed, multi-layer neural networks, sometimes called multi-layer perceptrons (MLPs), have the ability to learn intermediate representations. Thanks to their non-linearity, they can learn to extract low-level features in their first layers, and increase the complexity and the abstraction of the features across the layers. Finally, the last layers can behave like a classifier, which allows having a unique model to process images.

These models can be expressed as $f_{\text{ec}}(.;\Phi)$, where Φ is a set of parameters that can be optimized towards a specific goal by a learning algorithm. Φ can be optimized by minimizing an objective function f_{obj}. In a supervised task, this optimization step can be expressed as:

$$\Phi^* = \arg\max_{\Phi} \; f_{\text{obj}}(\mathcal{X}_{\text{train}}, \mathcal{X}_{\text{test}}; \Phi) \tag{2.6}$$

where Φ^* are the parameters returned by the learning algorithm.

The back-propagation (BP) technique is largely used to train multi-layer networks. BP allows the efficient computation of the gradient of all the operations of the network thanks to the chain rule formula. Then, those gradients are used by an optimization method to minimize a loss function f_{obj}. This metric gives the error between the actual predicted value and the expected value. Gradient descent (GD) is an optimization algorithm which uses all the training examples to update the parameters. When used with suitable values of the meta-parameters (e.g. a small learning rate), this method may find a smaller or equal loss after each step. However, this method has the disadvantage of being very expensive to compute and it can get stuck in local minima. Stochastic gradient descent (SGD) is another optimization method that uses only one image per update step, which reduces the risk of getting stuck in local minima by constantly changing the loss f_{obj}. However, SGD gives a stochastic approximation of the cost gradient over all the data, which means that the path taken in the descent is not direct (i.e. a zig-zag effect happens). Finally, a compromise exists between the SGD and GD methods: it averages the updates of multiple samples (defined by the batch size) to improve the robusness of SGD without the drawbacks of GD.

Finally, mini-batch gradient descent is a compromise between the two previous methods, using a subset of samples per update.

ANNs, especially deep learning ones, require large amounts of annotated data to be trained. This issue can be mitigated by the use of unsupervised learning models. Unsupervised representation learning is recognized as one of the major challenges in machine learning [55] and is receiving a growing interest in computer vision applications [56]–[58].

Such models can also be used with unsupervised learning, where labels are not available. In this learning mode, one tend to learn f_{e} instead of f_{ec}. The optimization problem becomes:

$$\Phi^* = \arg\max_{\Phi} \; f_{\text{obj}}(\mathcal{X}_{\text{train}}; \Phi) \tag{2.7}$$

In this case, f_{obj} cannot be formulated towards a specific application. Instead, some surrogate objective must be defined, that is expected to produce features that can fit the problem to be solved. Examples include image reconstruction [59], image denoising [60], and maximum likelihood [61]. In some cases, learning rules are defined directly without formulating an explicit objective function (e.g. in k-means clustering [56]).

One of the most used unsupervised ANN are the auto-encoders (AEs) [59], [62]. Instead of using an expected value that relies on the class of the data, the network is trained to reconstruct its input, and so, the loss function minimizes the difference between each input image and the reconstruction of this image at the output of the network. To do so, the topology of the network is divided into two parts: the first part, the encoder f_{enc}, projects the input into a smaller feature space. Then a decoder f_{dec} does the inverse transformation, by projecting this hidden state back into the input space. By using a smaller space for the intermediate representation, the network is constrained to compress the data into an efficient representation in order to be able to correctly reconstruct the input. Using higher feature space is also possible, but additional constraints need to be added to the model [56], [63].

Other models of neural networks exist. For example, restricted Boltzmann machines (RBMs) use the contrastive divergence (CD) method to learn a probability distribution of the input by alternatively sampling the input (called the visible layer) and the output (hidden layer) [64]. RBMs can be stacked, similarly to multi-layer neural networks, to learn more abstract features; such a network is called deep belief network (DBN) [61]. One of the limitations of neural networks, notably the deep architecture, is the huge amount of data required to learn a model which is able to generalize well. However, some datasets do not provide enough samples. One solution is to use data augmentation, by generating new samples by the deformation of the available samples.

2.1.4 Object Recognition Datasets

Since most of the models are trained on a sample collection to infer a generalization, a particular attention should be given to the data. For example, if the complexity of the data is too high according to the number of available samples, finding a correct model is hard. Also, if the data are biased, the trained model has serious risks to learn this bias and so to generalize poorly. Nowadays, gathering a huge number of various images is no longer a real problem, thanks to the search engines and social networks. However, getting correct labels still requires efforts in the case of supervised learning. Labeling can be a laborious work since it requires experts on complex tasks, and it needs to be done by hand to prevent as much as possible errors.

Multiple datasets, with different specificities, exist to compare the different methods used in image classification. While some datasets try to constrain the images in order to limit the variations and so, the classification difficulty, other datasets aim to be very heterogeneous in order to better represent in the real-world contexts. The number of samples available for each class is an important criterion since using more images allows improving the generalization of the model. Thus, trivial datasets, (i.e. toy datasets) can be limited to a few hundreds or a few thousands of images, while more complex datasets generally contain millions of images. A second important criterion is the number of classes. Working with more classes tends to make the problem more difficult. Some datasets contain only a set of object classes while others contain a hierarchy of classes, with abstract concepts gathering multiple object classes. While some datasets provide only the label associated with each image, others provide more information, such as a multiple keywords, bounding boxes of the different objects present in the images, or the segmentation of the pixels belonging to the objects.

An example of a non-challenging dataset is Modified-NIST (MNIST) [65]. MNIST consists of 60,000 training samples and 10,000 test samples. Each image is a grayscale, centered, and scaled handwritten digit of 28×28 pixels, which limits the variations to be taken into account (see Figure 2.6). The dataset has 10 classes, which are the digits from 0 to 9. Variants of this dataset exist, to test models on different properties. As an example, a permutation-invariant version exists (PI-MNIST) which prevents the usage of the spatial relationships between the pixels. Sequential-MNIST is another variant which consist to get one pixel at time [66]; it is notably used in recurrent approaches in order to evaluate the short-term memory of the network. NORB [67] and ETH-80 [68] are other toy datasets, which provide a few images of some objects. Again, the variations are limited (all the objects are

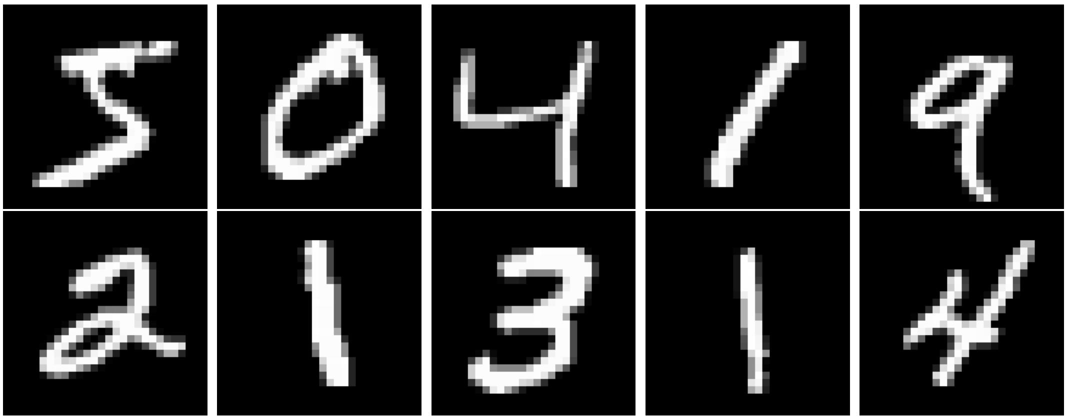


Figure 2.6: Samples from MNIST.

nearly centered and scaled, lighting conditions are good...). Only a few tens or hundreds of samples are provided, which is not really a difficulty because of the low complexity of these datasets. This kind of dataset is no longer used nowadays in the computer vision literature, because recognition rates are already very close to the maximum reachable (e.g. 99.82% on MNIST, see Table 2.1). The range of applications with such data remains also limited, since models trained on them work correctly only if the same constraints are present. However, these datasets are still useful for testing methods that are ongoing research efforts, like those detailed in the rest of this manuscript, because it limits the difficulty and allows to quickly prototype before starting to work on more complex tasks.

However, the advances of image classification methods have made it possible to focus on more complex tasks. These models aim to work on unconstrained data in order to be effective in most situations. But, in order for the models to be able to successfully generalize to new data, datasets that are intended for this purpose need to provide many examples for each class. As an example, Caltech-101 [69] and Caltech-256 [70], offer more diversity in images than previous datasets, but do not offer a sufficient number of images (i.e. 9,144 and 30,607) to effectively train current models. Recent datasets, such as ImageNet [71], Open Image [72], or MS-COCO [73] answer this issue by providing millions of labeled and unconstrained images (see Figure 2.7). These datasets are currently heavily studied because they offer real-world challenges and also enough samples to solve them. In addition, complementary information is provided. For example, ImageNet uses a vast hierarchy of concepts as classes, which allows gathering similar objects and testing the scalability to large class sets. Both ImageNet and Open Image provide the bounding boxes of the objects. MS-COCO gives the segmentation for each object in the images. Testing methods on such datasets is interesting in order to evaluate the performance of models in very complex tasks, but also their scalability. However, such datasets require very large computation times, both due to the high number of samples and the size of images, which is not adapted for most of the methods [74].

CIFAR-10 and CIFAR-100 [75] are also challenging datasets, still in use today. Although the number of training images is quite limited (50,000), the main challenges come from the low resolution of the images (32×32) and the large intra-class variations (see Figure 2.8). These datasets are an interesting compromise between the challenge of the task, and the computational power required to process them.

Some datasets are intended for unsupervised learning. This is the case of STL-10, which is built from ImageNet [56]. These datasets provide a large number of unlabeled data that can be used to learn features in an unsupervised way, and a small subset of labeled examples, that are used to train the classifier. Since the amount of training labeled examples is rather small (5,000 for STL-10), models

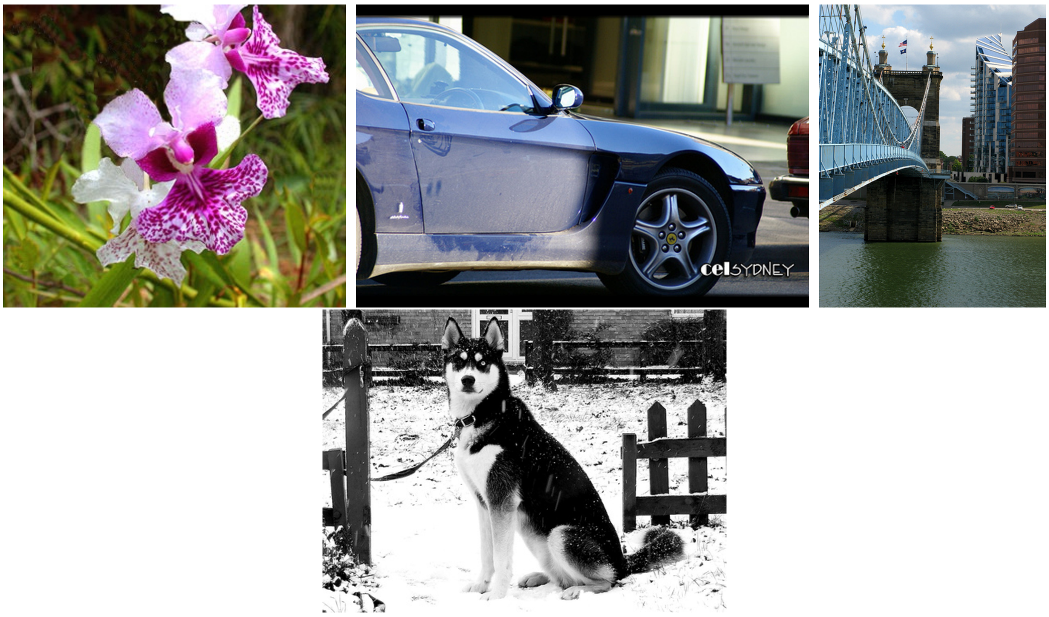


Figure 2.7: Samples from ImageNet.

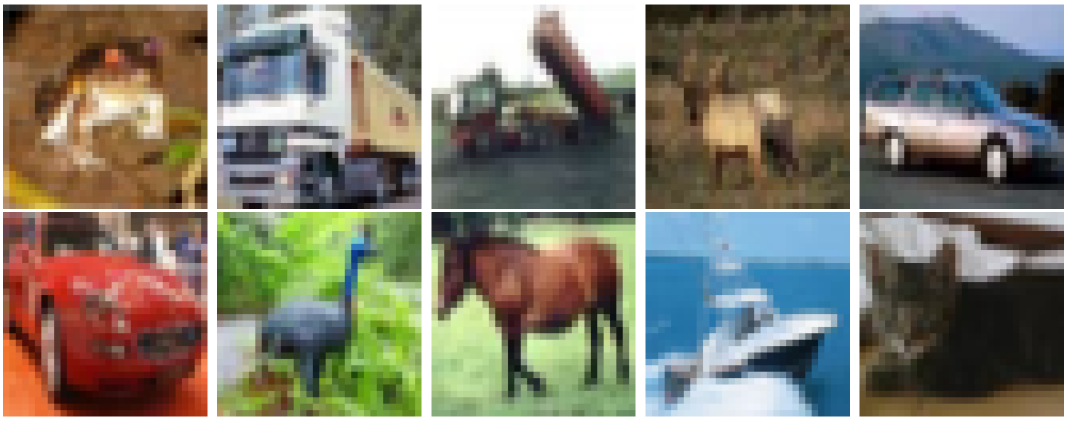


Figure 2.8: Samples from CIFAR-10.

Table 2.1: State-of-the-art performances on the different datasets. Deep Convolutional neural networks (CNNs) are the state-of-the-art method on most of the datasets.

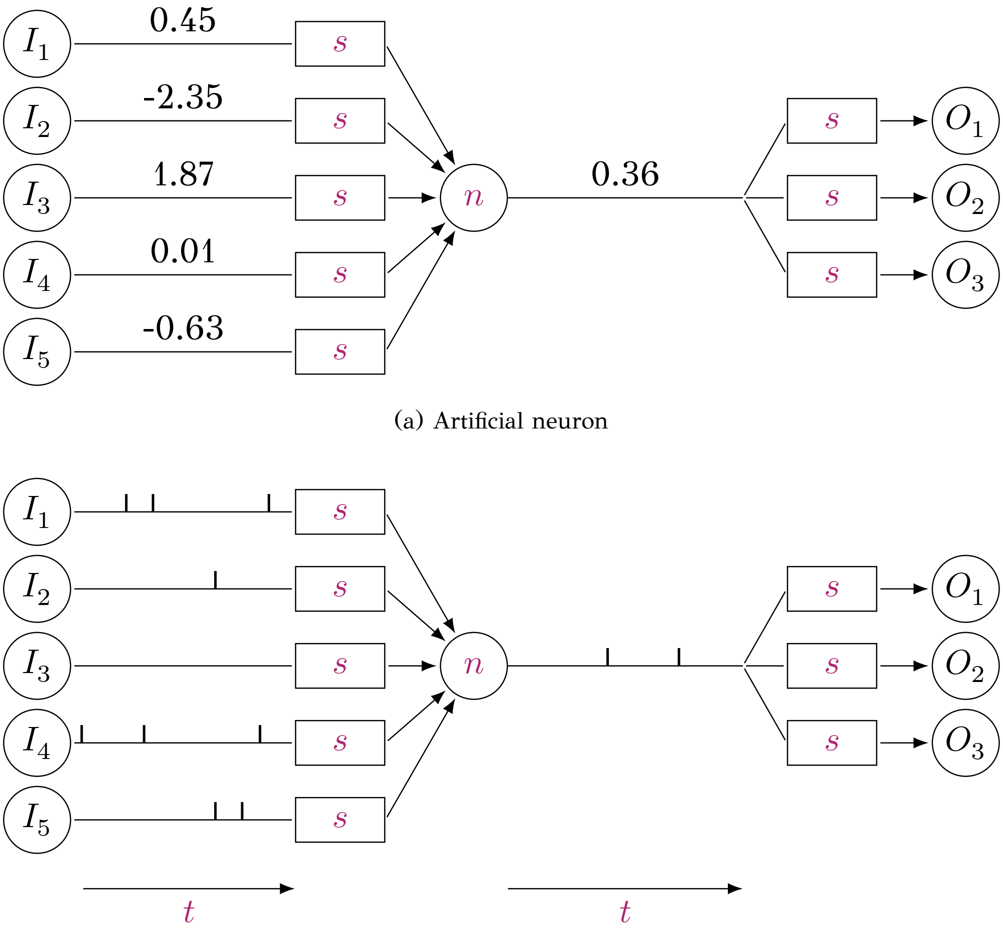
Dataset	Author	Method	Recognition Rate
MNIST	Kowsari et al. (2018) [76]	Deep CNN	99.82%
PI-MNIST	Pezeshki et al.(2016) [77]	Augmented MLP	99.43%
Sequential-MNIST	Cooijmans et al. (2016) [78]	LSTM	99.00%
CIFAR-10	Huang et al. (2018) [74]	Deep CNN + transfer learning	99.00%
SVHN	Zhang et al. (2019) [79]	Deep CNN	98.70%
ETH-80	Hayat et al. (2015) [80]	AE+ CNN	98.25%
NORB	cirecsan et al. (2011) [81]	CNN	97.47%
Caltech-101	Mahmood et al. (2017) [82]	Deep CNN + transfer learning	94.70%
CIFAR-100	Huang et al. (2018) [74]	Deep CNN + transfer learning	91.30%
ImageNet	Huang et al. (2018) [74]	Deep CNN	84.30% (Top-1 error)
Caltech-256	Mahmood et al. (2017) [82]	Deep CNN + transfer learning	82.10%
STL-10	Danton et al. (2016) [83]	Deep semi-supervised CNN	77.80%

need to learn a good intermediate representation on the unlabeled subset (100,000 for STL-10) to get good performance on the test set.

A summary of the datasets for object classification is given in Table 2.1.

2.2 Overview of Spiking Neural Networks

Despite their state-of-the-art performances on multiple tasks, ANNs also have some drawbacks. One of the most problematic is the energy efficiency of the models. As an example, deep neural networks require hundreds or even thousands of Watts to run on a classic architecture. Even tensor processor unit (TPU), which are optimized to simulate neural networks, consume about a hundred Watts [84]. In comparison, the brain uses about 20 W as a whole. A second issue is the supervision. Current unsupervised methods are far behind the capacity of the brain. Studies of models of an intermediate abstraction level, between the precise biological neural networks and the abstract artificial neural networks, aim to overcome these limitations. This family of neural network models, SNNs, uses a mode of operation closer to biology than ANNs, in order to benefit from its advantages, while allowing a simpler implementation [85], [86]. The main difference between ANNs and SNNs is their mode of communication. ANNs behave like a mathematical function: they transform a set of input numerical values into another set of output numerical values (see Figure 2.9a). Although this model of operation can be easily implemented on von Neumann architectures, the constraints of such models, like the need for synchronization, make them difficult to be efficiently implemented on dedicated architectures. In contrast, SNNs use spikes as the only communication mechanism between network components (see Figure 2.9b). These spikes, whose principle comes directly from biology, allow a complete desynchronization of the system, because each component is only affected by the incoming spikes. Depending on the model, each spike can be defined by a set of parameters. In its simplest form, a spike can be considered as a binary event, which means that the intensity or the shape of the impulse is neglected. Thus, the only parameter is t, the timestamp of the spike. A second parameter, the voltage v_{exc}, can be added to define a spike in some models. However, using spike computation prevents the usage of traditional learning methods, which are value-based. New methods need to be introduced in order to train SNNs. Despite the fact that the performances in terms of classification rate of these models are currently behind ANNs, the theory shows that SNNs should be more computationally powerful than their traditional counterparts [87], which means that SNNs should be able to compete with ANNs.



(a) Artificial neuron

(b) Spiking neuron

Figure 2.9: Comparison between an artificial neuron and a spiking neuron. (a) An artificial neuron applies operations on a set of numerical values to compute an output numerical value. (b) A spiking neuron receives a set of input spikes, and generates a set of output spikes.

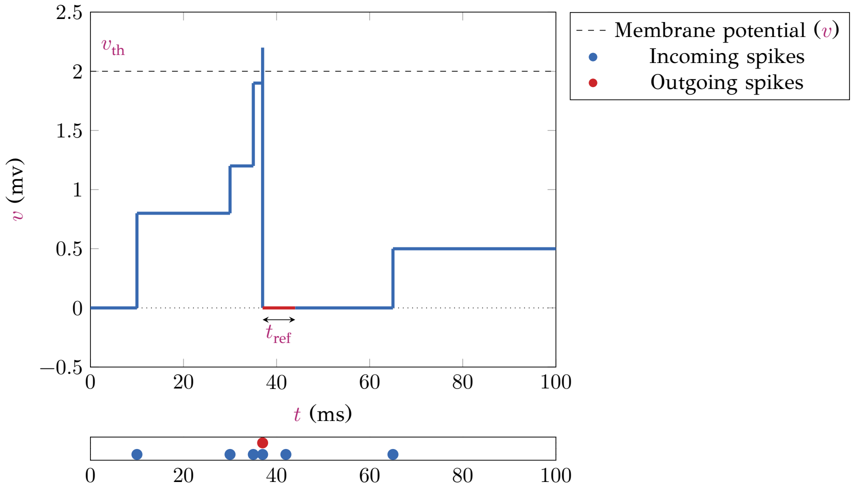


Figure 2.10: Evolution of the membrane potential of an IF neuron according to an incoming spike train.

2.2.1 Spiking Neurons

Spiking neurons are an intermediate model between biological neurons and artificial neurons. However, there is no consensus on the best trade-off between these two extremes. On the one hand, some models try to be biologically accurate, by sacrificing the simplicity of the model. On the other hand, other models are abstracted from biology by keeping only the most important principles, and thus, keeping a low computational complexity. So, there are multiple spiking neuron models, which offer different trade-offs [88]. This section presents some of the most used spiking neurons models. A spiking neuron is defined by its response to an input current z. In its simplest form, this current can be expressed from the input spikes as:

$$z(t) = \sum_{e \in \mathcal{E}} v_{\text{exc}_i} f_{\text{spike}}(t - t_i) \tag{2.8}$$

with $\mathcal{E}$ the set of incoming spikes, v_{exc_i} the voltage of the i^{th} spike, t_i the timestamp of the i^{th} spike and f_{spike} the kernel of spikes. A straightforward kernel is to use dirac impulses, denoted δ:

$$\delta(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.9}$$

More complex kernels exist, which allows to improve the level of details, such as the difference of exponentials ($f_{\text{spike}}(t) = e^{-\frac{t}{\tau_1}} - e^{-\frac{t}{\tau_2}}$).

Integrate and Fire model

One of the simplest models is called integrate-and-fire (IF) [89]. This model integrates input spikes to its membrane potential v. If v exceeds a defined threshold v_{th}, an output spike is triggered and v is reset to it resting potential v_{rest}. After firing, the neuron enters a refractory mode for the duration of t_{ref}. No spikes are integrated during this period (see Figure 2.10). The model is defined by the following formula:

$$c_{\text{m}} \frac{\partial v}{\partial t} = z(t), v \leftarrow v_{\text{rest}} \text{ when } v \geq v_{\text{th}} \tag{2.10}$$

with c_{m} the membrane capacitance.

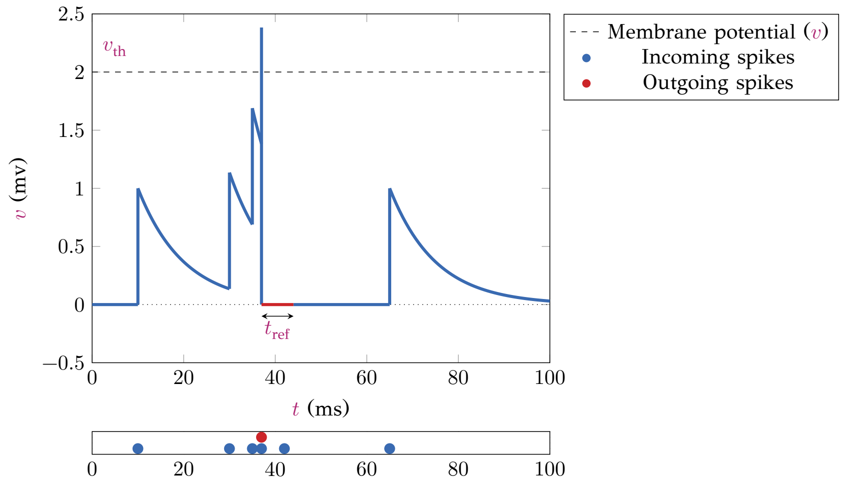


Figure 2.11: Evolution of the membrane potential of a LIF neuron according to an incoming spike train.

Leaky Integrate and Fire model

Leaky integrate-and-fire (LIF) models are a little bit closer to biology, by adding a leak to the membrane potential v. This leak allows neurons to return to the resting state in the absence of activity (see Figure 2.11). LIF can be expressed as:

$$\tau_{\text{leak}} \frac{\partial v}{\partial t} = [v(t) - v_{\text{rest}}] + r_{\text{m}} z(t), v \leftarrow v_{\text{rest}} \text{ when } v \geq v_{\text{th}} \tag{2.11}$$

with $\tau_{\text{leak}} = r_{\text{m}} c_{\text{m}}$ the time constant that controls the shape of the leak and r_{m} the membrane resistance.

There are more complex models belonging to the IF neuron family, such as exponential integrate-and-fire (EIF), quadratic integrate-and-fire (QIF) and adaptive exponential integrate-and-fire (LIF) neuron models which allows to achieve behaviours closer to biological observations.

Izhikevich's model

More complex spiking neurons exist, such as Izhikevich's. This model has the advantage of being relatively simple, but allows reproducing many of the firing modes observed *in vivo* [90].

$$\begin{aligned} \frac{\partial v}{\partial t} &= 0.04v^2 + 5v + 140 - U + z(t), v \leftarrow c, U \leftarrow U + d \\ &\text{when } v \geq 30 \text{ mV} \\ \frac{\partial U}{\partial t} &= a(bv - U) \end{aligned} \tag{2.12}$$

with a, b, c, d the parameters that set the firing mode of the neuron [90].

Hodgkin-Huxley model

The Hodgkin-Huxley model is important in neuroscience, because it is very close to biology [91]. It uses four equations and tens of parameters that reproduce the behavior of different ions channels in natural neurons. However, this model is one of the most complex ones to simulate and requires a high number of operations [88], which prevents its usage in large scale SNNs [92].

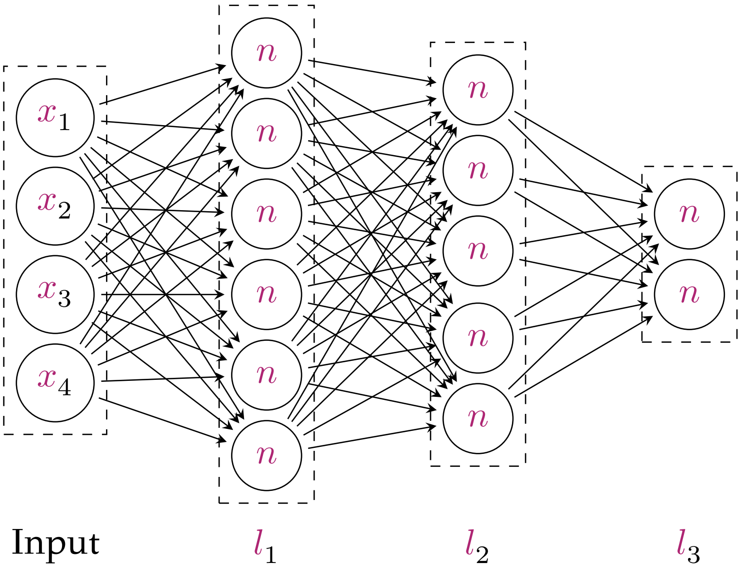


Figure 2.12: Example of a FF topology.

2.2.2 Topology

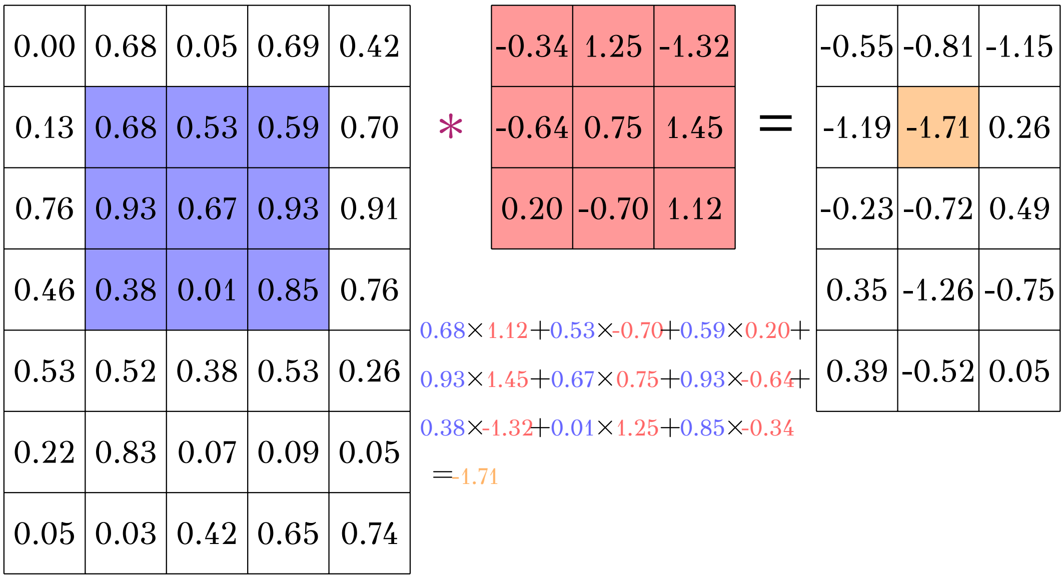
Neurons need to be connected to other neurons in order to create a network capable of performing the desired tasks. A connection is one-way, from an input neuron to an output neuron. The pattern of connections inside the network is called the topology. Basically, a neural network is a set of neuron $\mathcal{N} = \{n_0, n_1, \cdots, n_i\}$, connected by a set of synapses $\mathcal{S}$, with each synapse s_{ij} connect an input neuron n_i to an output neuron n_j. In the majority of topologies, neurons are gathered into groups, called layers $\mathcal{L} = \{l_0, l_1, \cdots, l_k\}$, each layer being defined as a set of neurons $l_i = \{n_0, n_1, \cdots, n_m\}$.

Feedforward Networks

In feed-forward (FF) topologies, layers are sequentially ordered in such a way that neurons in a layer l_i can only project their connections to subsequent layers l_j, with $i < j$ (see Figure 2.12). This constraint ensures that no cycle is possible. In this topology, neurons in early layers react to simple patterns, whereas neuron in deeper layers tends to react to more abstract or complex features. Most of the time, fully connected layers (sometimes called dense layers) are used: in this case, neurons of a layer l_i are connected to all the neurons of layer l_{i+1}.

Convolutional Networks

A CNN in the case of image processing, takes advantage of the nature of the data to improve its performances. In natural images, adjacent pixels have a strong covariance, which allows applying local filters to extract information. A CNN is a FF network with specific layers and connection patterns (see Figure 2.14). The convolution operation (see Figure 2.13) computes the amount of overlap between two functions and is expressed as: $\mathbf{A} * \mathbf{B}$. In the case of images, the functions are the image and the image features. Thus, a convolution layer is defined by a set of n trainable filters $\mathcal{F} = \{h_0, h_1, \cdots, h_n\}$ of size $h_{\text{width}} \times h_{\text{height}}$ (also called kernels). For an input layer of size $l_{\text{width}}(i-1) \times l_{\text{height}}(i-1) \times l_{\text{depth}}(i-1)$, a convolution layer will have a set of n feature maps of size $l_{\text{width}}(i) \times l_{\text{height}}(i)$, following the equation:



0.00	0.68	0.05	0.69	0.42
0.13	0.68	0.53	0.59	0.70
0.76	0.93	0.67	0.93	0.91
0.46	0.38	0.01	0.85	0.76
0.53	0.52	0.38	0.53	0.26
0.22	0.83	0.07	0.09	0.05
0.05	0.03	0.42	0.65	0.74

$*$

-0.34	1.25	-1.32
-0.64	0.75	1.45
0.20	-0.70	1.12

$=$

-0.55	-0.81	-1.15
-1.19	-1.71	0.26
-0.23	-0.72	0.49
0.35	-1.26	-0.75
0.39	-0.52	0.05

Figure 2.13: The two dimensional discrete convolution operation.

$$
\begin{aligned}
l_{\text{width}}(i) &= \frac{l_{\text{width}}(i-1) + 2l_{\text{pad}} - h_{\text{width}}}{l_{\text{stride}}} + 1 \\
l_{\text{height}}(i) &= \frac{l_{\text{height}}(i-1) + 2l_{\text{pad}} - h_{\text{height}}}{l_{\text{stride}}} + 1 \\
l_{\text{depth}}(i) &= n
\end{aligned}
\tag{2.13}
$$

with l_{pad} the padding (i.e. pixels added at the border to increase the output dimensionality) and l_{stride} the stride (i.e. the offset between the convolution position). Convolution layers preserve the dimensionality of the data and allow to reduce the number of trainable parameters (i.e. synaptic weights) when using shared weights. In opposition to dense layers, neurons in convolution layers are connected only to a subset of the neurons of the previous layer. Each neuron is connected to $h_{\text{width}}(n) \times h_{\text{height}}(n) \times l_{\text{depth}}(n-1)$ neurons of the previous layer, which form the receptive field of the neuron. Convolution layers can be mimicked with SNNs, by using the right connection policy. However, sharing the kernel on the different convolution positions impose to use non local operations or memory. Implementing such mechanisms on neuromorphic hardware is an issue.

In addition to convolution layers, pooling layers are also used in convolutional architectures to improve the spatial invariance, to add more non-linearity, but also to reduce the dimensionality of the data across the layers [65], which improve image recognition performances. In ANNs, multiple types of pooling exist, such as max pooling or sum pooling, depending on the neighboring operation used. For SNNs, both max pooling or sum pooling can be mimicked according to used models. Finally, one or several dense layers tend to be used the in last layers to act like a classifier.

Recurrent networks

Finally, some topologies are recurrent, which means that there are some cycles in the network (see Figure 2.15). Reservoir computing is a typical recurrent topology. It generally uses an input layer, followed by a reservoir and a readout layer. The reservoir contains a population of randomly connected neurons, which allows projecting the input into a higher dimensional space. Then, a linear classifier may be sufficient in the readout layer to learn the different states of the reservoir [93].

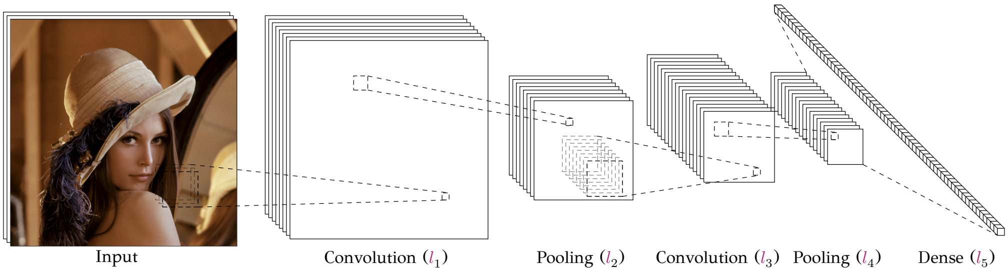


Figure 2.14: Example of CNN topology

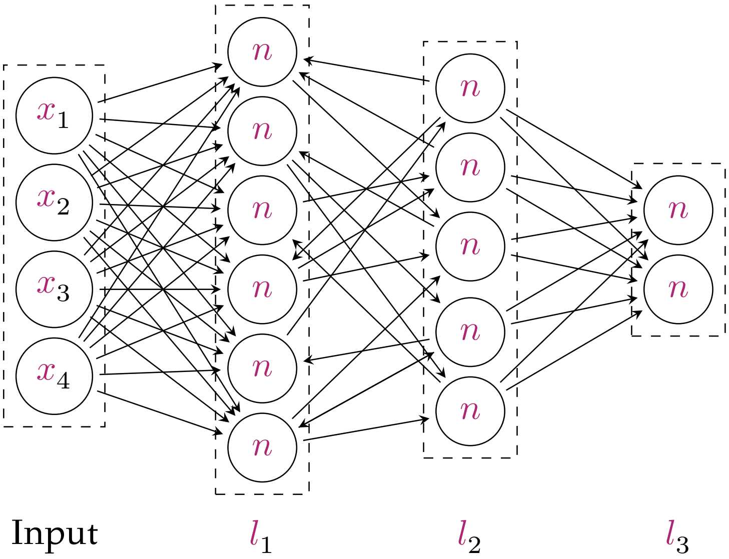


Figure 2.15: Example of recurrent topology

Long short-term memory (LSTM) is another typical topology, which uses memory cells to save states. Such topologies use gates which allow to control the impact of the previous inputs on the current input. Multiple models of LSTM exists with gates dedicated to forget, to memory, to select, or to ignore the data.

Recurrent networks are not studied in this manuscript, since processing static images does not require information about the past inputs. However, such a topology improves performance in applications in which the context is important over time, such as video processing or speech recognition.

2.2.3 Neural Coding

One of the most crucial mechanisms in SNNs is the neural coding [94]. The meaning of a spike, or of a population of spike, is an important question to address, since it plays a central role in the behavior of SNNs, and is related to the computational power of the network. Understanding the representation carried by a spike allows adapting the different mechanisms of the model to improve the performance of the network. Moreover, it is necessary to interpret the output of the network, which requires to associate values to the output spikes. In some applications, inputs are directly spike trains (e.g. with dynamic vision sensor (DVS) sensors), but if input data are not already coded as spikes, an input conversion function f_{in} is necessary to generate the spikes that will feed the network:

$$\begin{aligned} f_{\text{in}} : [0,1] &\to \mathbb{R}_+^{n_x} \\ x &\mapsto \left(t_0, t_1, \cdots, t_{n_x}\right) \end{aligned} \tag{2.14}$$

In opposition, an output conversion function f_{out} allows interpreting the output spikes:

$$\begin{aligned} f_{\text{out}} : \mathbb{R}_+^{N_x} &\rightarrow [0,1] \\ \left(t_0, t_1, \cdots, t_{N_x}\right) &\mapsto y \end{aligned} \tag{2.15}$$

These functions are directly related to the neural coding. Multiple neural codings are used in the SNN community, with different advantages, which leads to debates on which coding to use [95], [96]. In biology, the question of the neural coding is also not clear. Some studies suggest that multiple coding exists, and are used together in some areas of the brain [97].

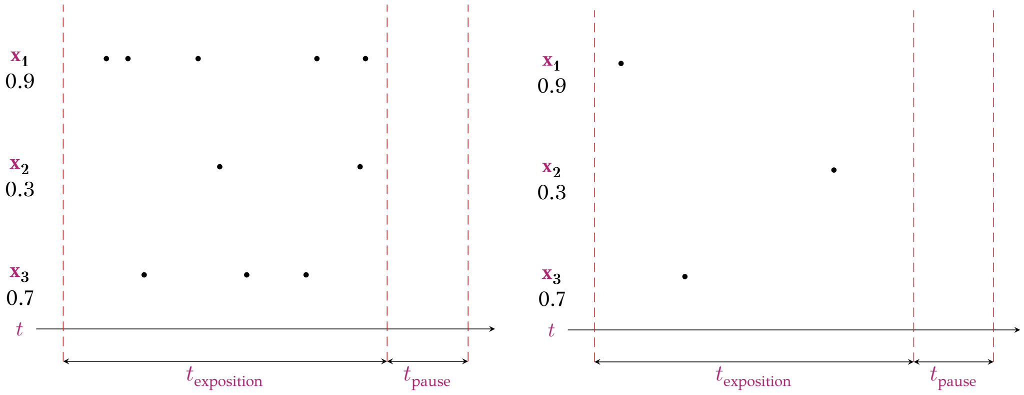
Frequency coding

Frequency coding [95], or rate coding, is one of the most used coding because it has been observed that some biological neurons emit spikes at a frequency proportional to the intensity of a stimuli (i.e. in the muscles [98], in the visual cortex [99]). Moreover, it is more straightforward to interpret the values in an ANN as a frequency of spikes. A common conversion function used to convert a numerical value into a spike train which respects frequency coding is the Poisson process: $t \sim \text{Poisson}(x)$, which can generate a series of discret events that occurs at a defined frequency with random timestamps. However, this coding has some limitations. To obtain an accurate estimate of the encoded values, large numbers of spikes need to be integrated by the neurons on each input connection. This means that the time constant must be large enough, or that the mean frequencies must be high. A related consequence is the introduction of latency into the network: each neuron needs enough time to integrate several spikes, in order to generate new spikes. Some authors show that such latency is not compatible with biological measurements, for example in the visual cortex [100]. However, some authors suggest that frequency coding allows reducing the noise in the system by averaging the activity over time [95].

In the case of static image processing, each sample needs to be exposed for a duration of $t_{\text{exposition}}$. The spike trains $(t_0, t_1, \cdots, t_{N_x})$ are generated by using the spike interval distribution according to the input value x. For $x \in [0,1]$, the spiking frequency F_{actual} is proportionally mapped to $[0, F_{\text{max}}]$. A pause duration t_{pause}, during which no spikes are generated, can be added in order to allow the neurons to return to a state close to their rest state (i.e. when the model of neurons includes a leak).

Temporal Coding

In opposition to rate coding, temporal coding assumes that information is directly carried in the timing of each spike. Therefore, one spike can be enough to represent an input value. Some work suggests that that the brain uses some kind of temporal coding [100], [101]. Different methods are used to generate spikes according to temporal coding. In latency coding [100], earlier spikes encode higher values, while later spikes represent lower values (see Figure 2.16b). The value that encodes a spike is the offset of the spike timing according to a time reference, which locates the beginning of the pattern. Similarly, rank-order coding [102] is another strategy which considers that the order of spike arrivals is more important than their exact timings. Input values are sorted and timestamps are generated based on the indices of the sorted values. These codings have the advantage of bringing more information with fewer spikes than rates coding. The latency issue, raised in frequency coding, is also reduced, since neurons can integrate only one spike per input before triggering an output spike. However, the system is more sensitive to noise because a time lag of a few milliseconds is enough to change the information carried by a spike.



(a) Frequency coding

(b) Temporal coding

Figure 2.16: Major neural codings.

Population coding

Some codings use several neurons to encode one value [103]. Such methods have the advantage of improving the encoding precision, but also of increasing the tolerance to noise by averaging the activity over a population of neurons.

Population coding, notably sparse population coding, seems to be used in the higher areas of the visual cortex [1]. Much work uses the grandmother cell approach in classification: a specific object activates one output unit. However, this design choice is poorly scalable, and requires as many output neurons as possible objects. Representing information across multiple neurons increases the amount of information carried and can reduce the noise of each neuron [104].

Phase Coding

Biological studies have highlighted the presence of oscillations in the brain, notably, delta (1-3 Hz), theta (4-8 Hz), alpha (~10 Hz), beta (15-25 Hz), and gamma (30-100 Hz) frequencies [105]. The different frequency of oscillations may help the synchronization of the neurons and play an important role in the brain [106]. Thus, based on these studies, some work uses the timing of spikes according to the background oscillations to code information [107].

2.2.4 Synapses

Alongside neurons, synapses are the second network component that plays a major role in SNNs. Synapses can be present on the connection between two neurons. Their role is to modulate the voltage of the spikes transmitted on this connection, in order to modulate the influence of the input neuron over the output neuron. The modulation factor is defined by the synapse weight w. A weak synaptic weight (i.e. close to zero) will greatly reduce the influence of the input neuron over the output neuron, since spikes that reach the output neuron will have a low voltage (i.e. behave the same way as if there was no spike at all). On the contrary, a strong weight will produce post-synaptic spikes with high voltages, which will significantly affect the output neuron state. Adapting the weight of the synapse in the network directly affect the pattern that will excite the neurons, and thus the tasks that the network is able to solve. Training a network consists notably to make this adaptation, thanks to learning rules. It is sometimes preferable, even necessary, to limit the weight range: $w \in [w_{\text{min}}, w_{\text{max}}]$. A delay d can be used as an additional synapse parameter. This parameter defines the delay added to each spike that passes through the synapse: $t_{\text{post}} = t_{\text{pre}} + d$.

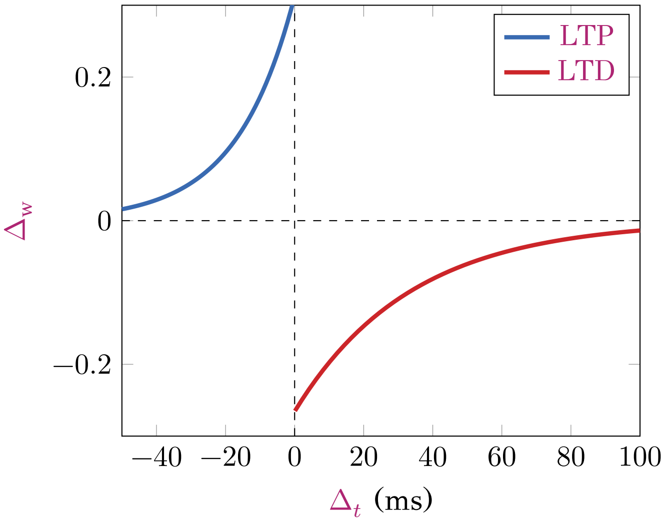


Figure 2.17: The biological STDP.

One of the first synaptic learning rule in ANNs was introduced by Hebb in 1949 by the sentence: "cells that fire together, wire together" [108]. The idea is to reinforce synapses between neurons that show correlations in their activation patterns. This rule can be written as:

$$\Delta_{\text{w}} = \eta_{\text{w}} x y \tag{2.16}$$

with η the learning rate, x the value of the input neuron activation and y the value of the output neuron activation.

For SNNs, one of the most studied learning rule is the spike-timing-dependent plasticity (STDP). This rule was observed by Bi and Poo (1998) [109], and follows the hebbian principle. STDP suggests that the synaptic connection depends on the difference of spike timestamps. They formalized the rule as:

$$\Delta_{\text{w}} = \begin{cases} \eta_{\text{w}} e^{-\frac{t_{\text{pre}} - t_{\text{post}}}{\tau_{\text{STDP}}}} & \text{if } t_{\text{pre}} \leq t_{\text{post}} \\ -\eta_{\text{w}} e^{-\frac{t_{\text{post}} - t_{\text{pre}}}{\tau_{\text{STDP}}}} & \text{otherwise} \end{cases} \tag{2.17}$$

with η_{w} the learning rate and τ_{STDP} the time constant that controls the leak, t_{pre} and t_{post}, respectively the timestamp of fires for input and output neurons. This rule combines two mechanisms: the long-term potentiation (LTP), when the input neuron fires just before the output neuron, and the long-term depression (LTD) in the other case (see Figure 2.17).

2.2.5 Inhibition

Finally, another important mechanism is the inhibition. In biology, the population of neurons can be divided into excitatory neurons, which constitute about 80% of the population, and inhibitory neurons, which are the remaining 20% [9]. Inhibitory neurons act the opposite way to excitatory neurons: inhibitory spikes will decrease the action potentials of the output neurons. In SNNs, this mechanism is also used, but with more degrees of freedom. In some work, neurons can both have output excitatory and inhibitory connections [28]. A frequent case of use of inhibition is competition: when a neuron reacts to a pattern, it sends inhibitory spikes to other neurons in competition to prevent them from firing, and thus, increase the sparsity of the activity. Pushed to the limit, this mechanism can be used to produce a winner-takes-all (WTA) policy: only one neuron can fire at once.

2.2.6 Homeostasis

A critical mechanism to ensure reasonable performance is to guarantee the homeostasis of the system [110]. When performing unsupervised learning without homeostasis, some neurons can take advantage over the others and, therefore, fire more spikes than the others. This phenomenon is intensified when a competitive mechanism such as inhibition is used, because the winner prevents other neurons from firing and the network gets stuck in a state where a single neuron is active, since it reinforces its synaptic connections. To prevent such a positive feedback loop, mechanisms are necessary to ensure the stability in the network.

Leaky Adaptive Threshold

In SNN, a common way to maintain the homeostasis of the system is to adapt neuron thresholds [111]. One method to do so is to use a leaky adaptive threshold (LAT) [112], which defines a new threshold ${v_{\text{th}}}'$ from the standard threshold v_{th} and an adaptive term Θ as:

$$\begin{aligned} {v_{\text{th}}}'(t) &= v_{\text{th}} + \Theta(t) \\ \frac{\partial \Theta}{\partial t} &= -\frac{\Theta(t)}{\Theta_{\text{leak}}} + \Theta_{+} z(t) \end{aligned} \tag{2.18}$$

with Θ_{leak} the leak constant, Θ_{+} the additive factor, z the output current of the neuron, and $\Theta(t)$ the adaptive part of the threshold according to time. This allows the neurons to regulate their spiking frequencies so that no neuron can strongly dominate the others. However, this method requires to carefully set Θ_{leak} and Θ_{+}parameters to balance output frequency and classification rates. Generally, an exhaustive search is required to optimize these parameters [112]. Moreover, it is not clear whether LAT is a relevant mechanism in all cases.

Intrinsic Plasticity

Some observations *in vivo* suggest that neurons adjust their excitability according to their activity. Some work [113]–[115] offers mechanisms which follow this idea. For example, intrinsic plasticity can be applied to the LIF model from [115]:

$$\begin{aligned} r_{\text{m}} &= r_{\text{m}} + \eta_1 \frac{2F_{\text{actual}}\tau_{\text{leak}}v_{\text{th}} - w - v_{\text{th}} - \frac{1}{F_{\text{expected}}}\tau_{\text{leak}}v_{\text{th}}{F_{\text{actual}}}^2}{r_{\text{m}}w} \\ \tau_{\text{leak}} &= \tau_{\text{leak}} + \eta_2 \frac{2t_{\text{ref}}F_{\text{actual}} - 1 - \frac{1}{F_{\text{expected}}}(t_{\text{ref}}{F_{\text{actual}}}^2 - F_{\text{actual}})}{\tau_{\text{leak}}} \end{aligned} \tag{2.19}$$

with r_{m} the resistance of the membrane, τ_{leak}, the membrane leak, η_1 and η_2 the learning rates, F_{expected} the desired mean of the output firing rate, F_{actual} the actual firing rate and t_{ref} the refractory duration.

Synapse Scaling

One way is to scale the weights of the incoming excitatory synapses of each neuron. Such methods prevent the summations of excitatory to give an advantage over the other neurons, which avoids entering into a feedback loop. It is possible to apply a global scaling, which ensures that all norms are strictly equals, and a local scaling, which is more biologically plausible and allows efficient hardware implementation [116]. Global scaling methods generally use the L1 norm [117]:

$$w_i = \frac{w_i}{\sum_j |w_j|} \tag{2.20}$$

Local synaptic scaling uses a multiplicative factor [118]:

$$\frac{\partial w_i}{\partial t} = \eta w_i (F_{\text{expected}} - F_{\text{actual}}) \tag{2.21}$$

with η the strength of the scaling, F_{expected} the target firing rates of the post-synaptic neuron, and F_{actual} an estimation of the actual firing rate.

BCM

Bienenstock-Cooper-Munro (BCM) is a model of synapses which uses a moving threshold to regulate the application of potentiation and depression according to the post-synaptic activity [119].

$$\tau \frac{\partial w}{\partial t} = F_{\text{pre}} F_{\text{post}} (F_{\text{post}} - \theta_{\text{th}}) \tag{2.22}$$

with F_{pre} and F_{post} the pre-synaptic and the post-synaptic firing rates, and θ_{th} the threshold to reach to apply LTD or LTP.

Short-term Synaptic Fatigue

Short term depression is a mechanism observed in biology [120]. Such a phenomenon can be used to prevent synaptic connections to strengthen to rapidly. The synapse efficiency, which is the ability to apply LTP, decreases when the frequency of incoming spikes increases [121]; this can modeled with a factor $G(t)$:

$$G(t) = w[1 - F(t)] \tag{2.23}$$

with w the synaptic weight and $F(t)$ a function which depends on the history of pre-synaptic spikes.

2.3 Image Recognition with Spiking Neural Networks

The behavior of SNNs differs from traditional methods. Specifically, in image recognition, some pre-processing is sometimes necessary to ensure the good behavior of SNNs (see Section 2.3.1). Moreover, different learning algorithms from ANNs seem necessary in order to train SNNs. Some work explore the conversion from traditional models to spiking models (Section 2.3.2). Other work focus on adapting traditional methods, such as back-propagation, in order to perform training directly in the spike domain (Section 2.3.3). Finally, some work studies bio-plausible rules, which allow to implement fully local, and sometimes unsupervised, learning rules (Section 2.3.4).

2.3.1 Pre-Processing

SNNs can require some pre-processing in order to achieve correct performances in image classification. A widespread method is on/off filtering, directly inspired from the bipolar cells situated in the retina. A straightforward method to reproduce the behavior of the biological retina is to use DoG filters [122]. Basically DoG filters can be defined as:

$$\text{DoG}(x, y) = \mathbf{X}(x, y) * (G_{\text{DoG}_{\text{size}}, \text{DoG}_{\text{center}}} - G_{\text{DoG}_{\text{size}}, \text{DoG}_{\text{surround}}}) \tag{2.24}$$

where $\mathbf{X}$ is the input image, $*$ is the convolution operator and $G_{K,\sigma}$ is a normalized Gaussian kernel of size K and scale σ defined as:

$$G_{K,\sigma}(u,v) = \frac{g_\sigma(u,v)}{\sum\limits_{i=-\mu}^{\mu}\sum\limits_{j=-\mu}^{\mu} g_\sigma(i,j)}, u,v \in [-\mu,\mu], \mu = \frac{K}{2}, \tag{2.25}$$

with g_σ the centered 2D Gaussian function of variance σ. The parameters of the filter are its size DoG_{size} and the variances of the Gaussian kernels $\text{DoG}_{\text{center}}$ and $\text{DoG}_{\text{surround}}$. Positive and negative values are generated following the application DoG filters. Thus, a second step is to separate these values into two channels: one for the positive values (i.e. representing the on cells), and the other for negative values (i.e. representing the off cells):

$$\begin{aligned} x_{\text{on}} &= \max(0, \text{DoG}(x,y)) \\ x_{\text{off}} &= \max(0, -\text{DoG}(x,y)) \end{aligned} \tag{2.26}$$

However it is not clear how to apply on/off filtering on color images, since only little work addressed this issue [7]. No work succeeds in managing to learn efficient features from colored images with an STDP learning to our knowledge. Biological studies report three opponent channels: black/white, red/green and yellow/blue [123]. Using similar channels can help to improve the processing of color images.

2.3.2 Artificial to Spiking Neural Networks Conversions

One of the most straightforward approaches to make an effective SNN that is able to process a defined task is to train an ANN with traditional methods (e.g. GD) on that task, and then convert the model into a spiking version. However, since the training is done offline, only the inference can benefit from the advantages of SNNs (i.e. the energy efficiency, or the low-latency responses). The main difficulty is to find a spiking network model that mimics as well as possible artificial network model, in other words, a conversion method that minimizes the error between the activities of the two models.

Early work addressed DBN conversion [124]. Since DBNs use binary activation units, it is straightforward to reproduce this behavior with spiking neurons: units with an output value of 1 should correspond to a neuron that fires, while the other neurons (i.e. a value of 0) should not fire. Similarly to a DBN unit that has a probability to be activated, it is possible to express the probability that a spiking neuron emits a spike. In [124], LIF neurons are shown to be an equivalent of Siegert units (i.e. a spiking neuron model) when the fire rate is normalized. This method allows to achieve performances close to the original model (i.e. 94.09% on MNIST for the spiking version against 95.2% for the artificial one). However, DBNs are not able to currently compete with neural networks trained with BP. Some work used probabilistic units in an ANN in order to facilitate the conversion [125]. Since the inputs, the synapses, and the units are all expressed as probabilities, it is possible to find spiking models that approximate the artificial model. By using only binary weights and bias in the converted spiking network, and by running it on the TrueNorth architectures, a performance of 99.42% is achieved on MNIST. Using ternary weights, [126] succeed in reaching 89.32% on CIFAR-10 and 65.48% on CIFAR-100. However, using binary units does not allow to be as accurate as continuous units. For this reason, some work offers conversion methods to transform continue units into a spiking approximated version.

A first difficulty with continuous units is negative values: in an artificial neural network, input values, weights, and bias, but also activation function outputs, can

Table 2.2: Performances of ANN-to-SNN conversion methods.

Dataset	Authors	Method	Recognition rate
MNIST	Rueckauer et al. (2017) [129]	CNN conversion	99.44%
	Esser et al. (2015) [125]	CNN conversion	99.42%
	Diehl et at. (2015) [127]	CNN conversion	99.10%
	Diehl et at. (2015) [127]	MLP conversion	98.60%
	Hunsberger et al. (2015) [128]	AE conversion	98.37%
	O'Connor (2013) [124]	DBN conversion	94.09%
N-MNIST	Stromatias et al. (2017) [130]	Classifier conversion	97.23%
CIFAR-10	Rueckauer et al. (2017) [129]	CNN conversion	90.85%
	Esser et al. (2015) [126]	CNN conversion	89.32%
	Hunsberger et al. (2015) [128]	CNN conversion	82.95%
	Cao et al. (2015) [7]	CNN conversion	77.43%
CIFAR-100	Esser et al. (2015) [126]	CNN conversion	65.48%
ImageNet	Rueckauer et al. (2017) [129]	CNN conversion	74.60%

be negative, which is not easily transformed into a spiking equivalent. A solution is proposed in [7], which uses the absolute value to prevent inputs to be negative, discard the usage of bias, and uses rectified linear unit (ReLU) activation functions ($\max(0, x)$) to avoid negative output values. In this way, ReLU activation can be approximated by LIF neurons. Since CNNs lead to state-of-art-performance on many object recognition datasets, [7] gives indications in order to get a spiking version of CNNs. Notably, it is necessary to use a sum-pooling layer instead of max-pooling layer in order to mimic their behavior with LIF neurons. This work succeeds in reaching 77.43% on CIFAR-10 with a spiking CNN, and shows that the approximation due to the conversion results only in small loss in performance (i.e. original CNN yields 79.12%). Other work suggests improvements, such as weight normalization in order to reduce the approximation error [127], the usage of a smooth LIF model combined with noise injection to improve results [128], and the introduction of spiking equivalents of well-used operations in ANNs (i.e. batch normalization, max-pooling, and softmax layers) [129]. [130] introduces a protocol to train an ANN on spiking stimuli, and then convert it into a SNN to directly process these stimuli. This is done by building histograms based on the spike trains in order to get analog vectors and then, train the ANN, and finally transfer the weights to a SNN. The results obtained by these authors are reported in Table 2.2.

2.3.3 Adapted Back-propagation

In opposition to ANN-to-SNN conversion methods, some work proposes to apply to spiking networks training methods similar to the ones used with ANNs. These methods do not suffer from the performance loss due to the approximation introduced in the ANN-to-SNN conversion process. However, it is not possible to directly use these methods in SNNs since they are not adapted to spikes. Since BP is well-used to train ANNs, many authors focus on adapting this method to work directly in the spike domain. Moreover, some work suggests that the brain can use mechanisms which mimic BP [6], [131], [132], which motivates the authors to find BP methods that are compatible with neuromorphic architectures.

Traditionally, BP is incompatible with SNN models for multiple reasons. The most important one is that back-propagation is not local in space and in time, which is a requirement to produce efficient hardware. In particular, some work raises the weight transport problem [133]: to propagate the error signal, a symmetry of the synapse weight is required in the feedback connection, because the gradient of the input is related to the weights. However, some work suggests that BP also works with asymmetric weights, like in [134], where random feedback weights are used.

Another issue is that spiking neurons are not differentiable [135], which prevents the computation of the gradients required to optimize the network. To bypass this issue, some work suggests using approximations of the behavior of spiking neurons in order to get a derivable function that can be used in the backpropagation process. This is done in [135], [136] by applying low-pass filters on the membrane potential of LIF neurons or in [137] by approximating IF neurons by a ReLU activation. Other work uses event-driven formulation [134], a combination of micro (by using post-synaptic potential (PSP) values) and macro (i.e. firing rate) information [138], adapt BP with temporal coding [139], or introduce spatio-temporal version of BP [140], [141].

BP also requires accurate error signals to correctly converge [135], which seems to be difficult to code in spike trains. However recent work suggests that approximating the signal is enough [142]. In [143], this signal is approximated with signed spikes, thanks to a method called spiking vector quantization. However this method requires numerous spikes to get an accurate value.

Moreover, defining a biology-plausible objective function is not straightforward [6]. Some work uses the AE architecture in order to train the network to reproduce the input spike trains [144]. Other work uses a delay error between firing timings of neurons and arbitrary timing objectives [145]. Finally, switching between forward and backward passes is not straightforward in the case of a continuous input [131].

The performances obtained with these methods are listed in Table 2.3. The interest of training the network directly on spikes can be shown by N-MNIST, the neuromorphic version of MNIST [146]. SNNs report better performance than ANNs (99.53% [141] vs 99.23% [147]) because inputs are already encoded as spikes (i.e. the conversion from spikes to values used in ANNs lead to an information loss). However, despite the good performance reported, none of these authors was successful in addressing all the problems previously raised (the locality of the computations, the alternance between forward and backward passes, the accurate propagation of the error signal...), which does not make it possible to implement such a model on neuromorphic architectures.

2.3.4 Local Training

Finally, much work focuses on fully local in space and time learning rules, in order to provide models that are both compatible with neuromorphic architectures and able to learn directly from spikes. Moreover, this work mostly uses unsupervised learning rules. One of the earliest work by Masquelier et al. [149] uses an HMAX architecture with a single layer (S2) trained with STDP. Authors use temporal coding and a radial basis function (RBF) network as classifier. They test the network on the motorbikes and the faces datasets of Caltech [1]. They tested the network in a face/non-face and motorbike/non-motorbike task, in which they reach 99.1% and 97.8% classification rates. They use a simplified STDP rule:

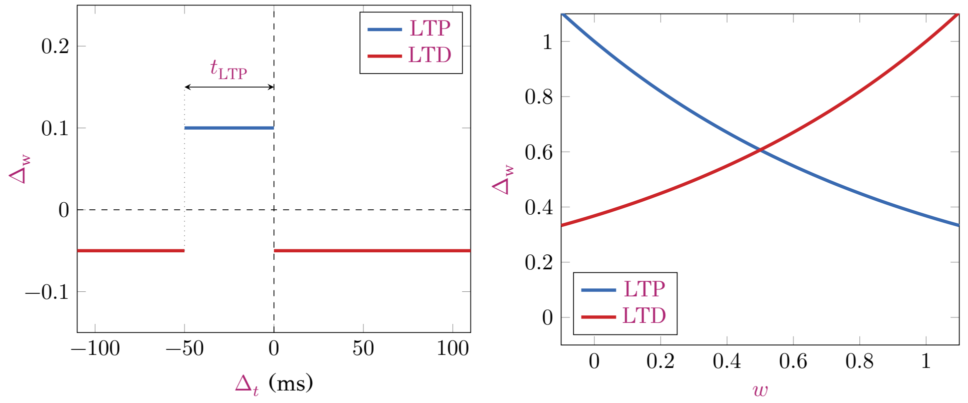
$$\Delta_{\mathrm{w}} = \begin{cases} \eta_{\mathrm{w}+} w(1-w) & \text{if } t_{\mathrm{pre}} \leq t_{\mathrm{post}} \\ -\eta_{\mathrm{w}-} w(1-w) & \text{otherwise} \end{cases} \tag{2.27}$$

with $\eta_{\mathrm{w}+}$ and $\eta_{\mathrm{w}-}$ the learning rates of LTP andLTD. Other work uses STDP with frequency coding. Querlioz et al. [111] use a multiplicative STDP rule in a single layer to achieve 93.5% on MNIST with 300 outputs units. This rule avoids the

[1] http://www.vision.caltech.edu

Dataset	Author	Method	Recognition Rate
MNIST	Lee et al. (2019) [136]	BP on LIF membrane potential (CNN)	99.59%
	Jin et al. (2018) [138]	Spiking BP	99.49%
	Wu et al. (2018) [140]	Spatio-temporal BP (CNN)	99.42%
	Lee et al. (2016) [135]	BP on LIF membrane potential (CNN)	99.31%
	Liu et al. (2017) [145]	Temporal BP (MLP)	99.10%
	Panda et al. (2016) [144]	Spiking AE BP	99.08%
	Tavanaei et al. (2018) [148]	Representation learning and BP (MLP)	98.60%
	O'Connor et al. (2016) [143]	Spiking BP (MLP)	97.93%
	Tavanaei et al. (2019) [137]	BP of IF neurons (MLP)	97.20%
PI-MNIST	Neftci et al. (2017) [134]	Random feedback BP (MLP)	97.98%
	Lee et al. (2016) [135]	BP on LIF membrane potential (MLP)	98.77%
	Mostafa et al. (2018) [139]	Temporal BP (MLP)	97.55%
N-MNIST	Wu et al. (2018) [141]	Spatio-temporal BP (CNN)	99.53%
	Lee et al. (2019) [136]	BP on LIF membrane potential (CNN)	99.09%
	Wu et al. (2018) [140]	Spatio-temporal BP (MLP)	98.78%
	Lee et al. (2016) [135]	BP on LIF membrane potential (MLP)	98.66%
CIFAR-10	Lee et al. (2019) [136]	BP on LIF membrane potential (CNN)	90.95%
	Wu et al. (2018) [141]	Spatio-temporal BP (CNN)	90.53%
	Panda et al. (2016) [144]	Spiking AE BP	70.16%

Table 2.3: Performances of adapted BP training methods.



(a) (b)

Figure 2.18: Multiplicative STDP rule ($\beta = 1$)

saturation effect by introducing a term that depends on the current weight (see Figure 2.18):

$$\Delta_w = \begin{cases} \eta_{w+} e^{-\beta \frac{w - w_{\min}}{w_{\max} - w_{\min}}} & \text{if } t_{\text{pre}} \leq t_{\text{post}} \text{ and } t_{\text{post}} - t_{\text{pre}} \leq t_{\text{LTP}} \\ -\eta_{w-} e^{-\beta \frac{w_{\max} - w}{w_{\max} - w_{\min}}} & \text{otherwise} \end{cases} \tag{2.28}$$

with β the parameter which controls the saturation effect (increasing β reduces the saturation). A particular form derived from this rule is the additive STDP rule, when $\beta = 0$.

Similarly, Dielh et al. [112] reach 95% with 6400 units and a power-law STDP. They use the synaptic trace r_{actual} that represents the recent history of spikes that go through the synapse:

$$\frac{\partial r_{\text{actual}}}{\partial t} = -r_{\text{actual}}(t) + \sum_{i \in S} \delta(t - t_i) \tag{2.29}$$

Thus, the update rule is:

$$\Delta_w = \eta(r_{\text{actual}} - r_{\text{expected}})(w_{\max} - w)^{\mu} \tag{2.30}$$

Dataset	Author	Method	Recognition Rate
MNIST	Kheradpisheh et al. (2018) [151]	Multilayered STDP	98.40%
	Tavanaei et al. (2016) [150]	Multilayered STDP (probabilistic rule)	98.36%
	Dielh et al. (2015) [112]	Single layer STDP	95.00%
	Querlioz et al. (2012) [111]	Single layer STDP	93.50%

Table 2.4: STDP training methods. Currently, these methods are mainly evaluated on simple datasets, such as MNIST.

with η the learning rate, r_{actual} and r_{expected} the actual and the expected spike trace of the synapse and μ the parameter to control the slope of the rule.

Recent work succeeds in using STDP to train several layers. Tavanaei et al. [150] learn convolution filters by a dedicated network, SAILNet, from patches extracted from input samples. A pooling layer and, then, a fully connected layer using probabilistic LIF neurons, are stacked. An SVM classifies the output of the last layer. This model reaches 98.36% on MNIST with 32 convolution filters and 128 output neurons. However, the usage of an external network to train convolutions remains an issue. Moreover, probabilistic LIF neurons are used in the feature discovery layer, which requires some global computation (softmax) to operate.

Kheradpisheh et al. [151] use two convolution layers trained by STDP and a temporal coding. The network reaches 98.4% on the MNIST dataset with 30 filters in the first convolution and 100 in the second convolution. However, this model uses some global computation: the potential of neurons is compared to each other to designate the winner at one time step and the filters are learned across the convolution columns. This model requires to tune its parameters carefully, especially the neuron thresholds. Moreover, the values of neuron thresholds must be manually changed between the training and the testing stages. Finally, the output neurons use infinite thresholds, which would not be realistic on hardware.

2.3.5 Evolutionary Algorithms

Evolutionary algorithms (EAs) are another family of algorithms inspired from biology, and in particular the theory of evolution. Some work uses EA in order to optimize SNN performances. Such algorithms can be used to directly optimize the synaptic weights w and delays d [152], or to optimize the network topology and the model hyperparameters [153], [154]. They can be an alternative to exhaustive search. However, EAs are very time consuming [155], notably because the fitness function is computationally expensive (i.e. simulating the performance of SNNs on a specified task). Thus, such methods are currently only applied to very simple tasks.

2.4 Software Simulation

Since producing dedicated hardware architectures is long and costly, using software simulation in the first place is an interesting choice to explore the different configurations. However, since neuromorphic systems are working very differently from von Neumann architectures, creating an efficient software simulator is challenging. To facilitate the work of the community, multiple simulation frameworks are available, each of them with different objectives. Some simulators focus on the level of detail, to provide an accurate reproduction of the models. Other simulators offer scalable solutions in order to be able to run large scale networks, often at the expense of the level of detail. It is possible to implement a neuromorphic simulator in two ways: by refreshing the models each tick of a clock or by updating the models at each time an event arises in the system. These two types of simulation will be discussed in Section 2.4.1. Then, a tour of the software simulators will be done in Section 2.4.2.

2.4.1 Event-Driven vs Clock-Driven Simulation

SNNs are essentially defined by standard differential equations, but, because of the temporal discontinuities caused by the spikes, designing an efficient simulation of spiking neural networks is a non-trivial problem [156]. There are two families of simulation algorithms: event-based simulators and clock-based ones. Synchronous, or clock-driven, simulation simultaneously updates all the neurons at every tick of a clock; it is easier to code, especially on graphical processing units (GPUs), for getting an efficient execution of data-parallel learning algorithms. Event-driven simulation behaves more like hardware, in which conceptually concurrent components are activated by incoming signals (or events).

Event-driven execution is particularly suitable for untethered devices such as neurons and synapses, since the nodes can be put into a sleep mode to preserve energy when no event is triggered. Energy-aware simulation needs information about active hardware units and event counters to establish the energy usage of each spike and each component of the neural network. Furthermore, as the learning mechanisms of spiking neural networks are based on the timings of spikes, the choice of the clock period for a clock-based simulation may lead either to a lower precision or to a higher computational cost [92].

There is also a fundamental difference between this event-driven execution model and the clock-based one: the event-driven execution model is independent of the hardware architecture of the computers on which it is running. So, event-driven simulators can naturally run on a cluster of computers, with the caveat of synchronization issues in the management of event timings.

2.4.2 Neuromorphic Simulators

A first group of software simulators in the neuromorphic community aims to reproduce the behavior of biological neural network as faithfully as possible. Well-used simulators in this category are NEURON [157] or GENESIS [158], which allow modeling multiple compartments in each neuron. However, such tools are able to simulate only few neurons, and so, do not allow working on neuromorphic architectures intended to solve complex tasks, which require several thousands of neurons generally. A second group gathers the clock-driven spiking neural simulators that are intended to simulate far more neurons by using simpler models. NEST [159], CarlSim [160], Brian [161], Nengo [162] belong to this category. Finally, the last group gathers event-driven spiking simulators, which use the sparsity offered by SNN in order to get more scalable. Examples of event-driven spiking simulators are Xnet [163] (i.e. recently integrated in N2D2 [164]) and SpikeNet [165].

An important feature provided by these simulators is the hardware architectures that can be used to run them. Basically, all simulators work on standard central processing unit (CPU) architectures. But some of the simulators bring GPU support, field-programmable gate array (FPGA) support, or dedicated architecture support. As an example, GPUs allow speeding up simulations by using single instruction multiple data (SIMD) processing, and thus, updating multiple units with a single instruction. However, this operating mode requires to use clock-driven simulation.

Simulators are also defined by the programming interface compatibility. One of the most used is PyNN [166], a python interface that allows describing neural network topologies.

2.5 Conclusion

Object recognition is a very active research field, due to its large range of applications. This field has been strongly impacted by the emergence of deep learning methods, which have greatly improved the performances of artificial methods on complex tasks (see Section 2.1). However deep neural networks have the disadvantage of being power hungry, which hampers their usage. SNNs are promising candidates to bypass this issue, because they allow highly energy efficient-architectures, especially when the activity inside the network is sparse (see Section 2.2). In return, such hardware imposes some constraints, such as the locality of computation and memory. Some work bypasses these requirements by proposing offline training methods (see Section 2.3.2). However, these models benefit from energy efficiency only during inference, and not during training. Other approaches use adapted BP rules to learn directly in the spike domain (see Section 2.3.3), but these models are not fully compatible with neuromorphic hardware. Finally, some learning rules match the requirement of neuromorphic architectures, such as STDP (see Section 2.3.4). However, these rules are currently immature, and so, do not allow to currently process complex tasks. This manuscript aims to propose new solutions to improve SNN performance on image classification tasks, in order to be able to process real-world datasets. Image classification has the advantage of decades of active studies and of a large number of dataset available. One of the main motivations throughout this manuscript is to provide multi-layered SNN models that allow learning from data with STDP.

The following of this manuscript focuses on improving the performance of SNNs on image classification tasks, while respecting as much as possible the constraints required by neuromorphic architectures. The aim is to enable the usage of SNNs on complex datasets. The first contribution focuses on the software simulation of SNNs (see Chapter 3) to provide tools that facilitate and accelerate the exploration of the models described in this manuscript. Then, Chapter 4 focuses on the frequency loss problem, which needs to be addressed to enable the creation of multi-layered SNNs. Chapter 5 investigates on the quality of the features learned with STDP on complex datasets, and compares STDP-based SNNs with AE. Finally, Chapter 6 proposes new mechanisms to make it possible to train multi-layered SNNs with STDP, and study the impact of the different mechanisms on the networks.

Contributions

Chapter 3

Software Simulation of SNNs

As discussed earlier, it is necessary to explore SNN models and parameters, because they are still immature. It is not possible to carry out this exploration directly on hardware architectures, since developing dedicated hardware is a long and expensive process. Thus, creating and using software simulators is a solution which speeds up the study of SNNs. It is much easier to modify mechanisms or parameters in a software simulator than in a hardware device.

However, creating SNN simulators is also challenging. It is necessary to design a simulator which meets a number of criteria. For instance, some simulators should be very flexible in order to facilitate their adaptation to a maximum of models and use cases, but, in return, they may lose efficiency due to overheads. Other simulators require to be scalable, for the purpose of running large networks. While some simulators tend to model as many details as possible to respect the biology or hardware fidelity, others use more abstract models to speed-up the simulation. There is a large number of criteria, and therefore, a significant number of SNN simulators have been developed to meet them according to the needs.

This chapter presents two SNN simulators developed in parallel of the studies presented in this manuscript. The first simulator is neural network scalable spiking simulator (N2S3) (Section 3.1), which aims to be flexible and scalable. The second simulator, the convolutional spiking neural network simulator (CSNNS) (Section 3.2), is design to be optimized to run specific SNN models.

3.1 N2S3

The neural network scalable spiking simulator (N2S3) [167] is an open-source simulator that is built to help the design of spiking neuromorphic circuits based on nanoelectronics [168], [169]. This simulator is used in Chapter 4. The creation of this simulator is motivated by multiple criteria. N2S3 should be scalable, in order to be able to run large networks. It must run in parallel, in order to take advantage of multi-core architectures, and so, be efficient, and should be distributable, to run simulations on a cluster when the size of a simulated network exceeds the capacity of a single computer. The second criterion is the flexibility. N2S3 should be as little specialized as possible, to allow implementing a large range of models (e.g. IF and LIF neurons), topologies (e.g. feed-forward and reccurent networks) and behaviors (e.g. delay, inhibition, weight sharing...). This criterion is required not to restrict the exploration of SNN models. Finally, N2S3 must be easy to use, in order to be usable by non-computer scientists. Since SNNs are an interdisciplinary research area which gather electronics, physics, biology, and neurosciences, software simulators should be taken in hand by the different communities.

This simulator is based on the Scala programming language [170], which offers major advantages: the mixture of the oriented-object and functional paradigms, the compatibility with the existing Java library, and the multi-platform support provided by the Java virtual machine (JVM). N2S3 is an event-driven simulator (see Section 2.4.1), which allows taking advantage of the time and space sparsity of SNNs in order to be scalable. In N2S3, the event-driven simulation relies on the actor model. Specifically, the Akka library [171] is used, because of its good Scala implementation, but also because it allows distributing the computations easily (so that the simulator can scale out a simulation on several computers to handle large networks). As a result, most entities of network simulations (neurons, synapses, inputs), but also most of the features of the simulator (visualization and measurement tools, reports), are contained in actors. N2S3 uses an abstraction layer to separate neural network modeling from actor distribution issues (see Figure 3.1) through *containers*: each network entity is contained in a container, each container may contain one or more entities, and each container corresponds to one actor. It provides complete control over the number of actors, and thus over the parallelism level of the simulation, independently of the topology of the simulated network. Each entity in the network has a uniform resource locator (URL), which allows to query them. Abstraction layers are built over this system in order to facilitate the creation of experiments. As an example, N2S3 uses *NeuronGroup*, which contains a set of neurons, and *ConnectionPolicy*, which contains a set of synapses, to help with the creation of the network. These entities automatically manage the creation, the deployment, the setting, and the destruction of the underlying actors.

Since actors are inherently concurrent, one concern is how the temporal order of messages is guaranteed during the simulation. To do so, N2S3 allows to configure several levels of synchronization to be used by the simulation designer. On one end of the spectrum, N2S3 may make use of a unique synchronizer for the simulation, which will ensure that no causality issues happen but can create a bottleneck that will affect the performance of the simulation. On the other end of the spectrum, N2S3 can be configured to use a synchronization mechanism which is local to each neuron. The latter policy enables better parallelism, but may cause some temporal consistency problems. Some work remains to be done in order to guarantee that no causality errors can happen without using a global synchronizer. A solution to this problem may be the implementation of parallel synchronizers in the parts of the network that do not contain cycles (e.g. in a FF network, one synchronizer per layer can ensure that no causality error arises).

The software is divided into several packages. One first distinction is made between the library part, which is the main part of the simulator, and the user part, in which the simulator can be extended with new models, simulations, and features. Within the library part, the core functions (i.e. minimal functionalities required to run the simulator) are separated from the optional features and from the basic models of neural networks.

N2S3 uses a piped stream system to provide stimuli to the network entities. The input process typically starts by an input reader, which reads data from files or any external source, followed by a number of streams that filter the input data before feeding it to the network. Input readers provided with N2S3 allow to read data in a variety of formats, including standard formats such as address-event representation (AER), a data format used by spike-based cameras, or MNIST, used in a standard dataset for handwritten digit recognition. Subsequent filter streams available in N2S3 include neural coding streams, which convert raw numerical data into sequences of spike timings, spike presentation streams (e.g., repeating input spikes over a given period, or shuffling spikes), and modifier streams, that alter the input spikes (e.g., by adding noise). Users are free to use one or multiple input readers and to combine any number of filter streams in any order; they may also easily create their own readers and filters.

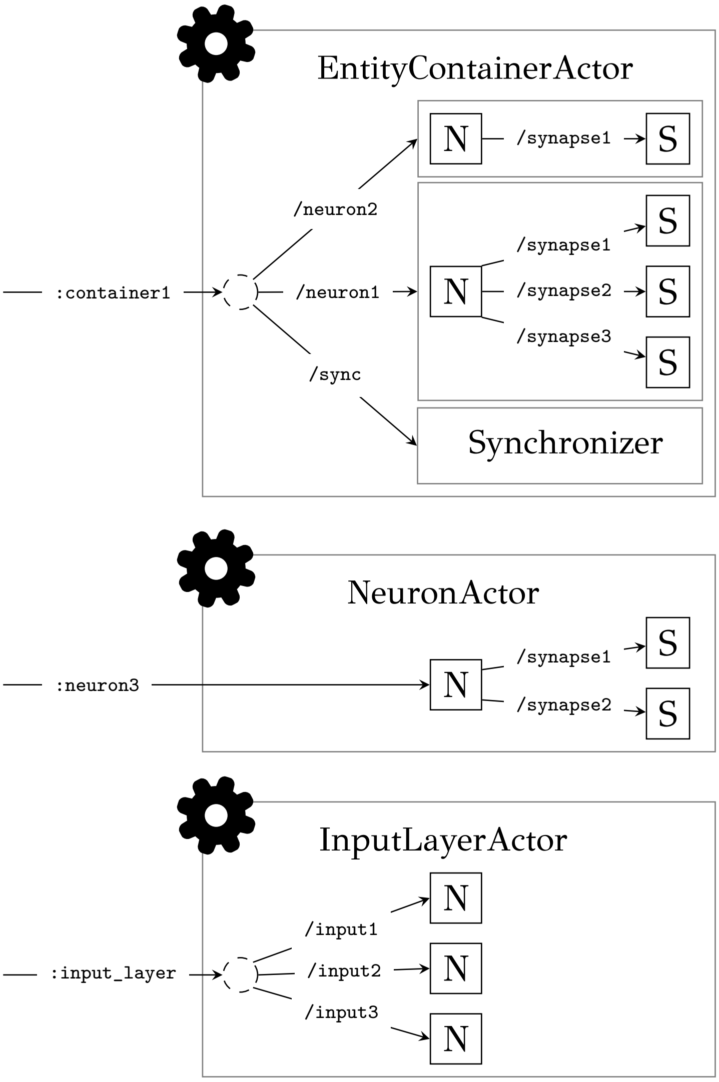


Figure 3.1: In N2S3, a network is organized in specialized actors that may contain one or more network entities. Such entities could be, for instances, neurons, inputs or any other. Each entity can be queried thank to its URL.

Table 3.1: Feature-wise comparison of SNN simulators: N2S3, NEST (Python) [159], Brian [172], and Xnet [163] (currently integrated to N2D2 [164]).

Feature	PyNEST	Brian	Xnet	N2S3
Topologies				
Feed Forward	X	X	X	X
Reccurent	X	X	?	X
Models creation				
Analytics		?	X	X
Differential equations		X	X	
Inputs				
Spike Generators	X	X	?	X
Temporal Coding			X	X
Others Features				
Energy consumption	X	?	X	X
Distributable	X			X

Table 3.2: Comparison of simulator performances on the same configuration (Ubuntu 14.04, i5 core, 4GB RAM). Results may vary between different runs due to the software stack.

Experiment	Measure	PyNEST	Brian	N2S3
MNIST, 100N	CPU time	15:05:16	9:39:15	3:42:03
	Memory	85 MB	2822 MB	1331 MB
Freeway, 60N	CPU time		10:03:40	3:34:41
	Memory		914 MB	1448 MB

Users may observe simulation outputs (spikes, weight values...) through network observers. Network observers follow the observer pattern by subscribing to events in the simulation (e.g., when a spike happens), perform some calculations on such events, and make them visible to the user. Examples of such observers range from textual loggers to dynamic visualizations of the spikes of each neuron. Concretely, N2S3 provides a spike activity map of the network, a synaptic weight evolution visualizer, and the calculation of evaluation metrics (e.g. recognition rates, confusion matrices...).

It is possible to use N2S3 directly with the Scala interface (see Appendix A.1). In addition, N2S3 includes a dedicated internal domain specific language (DSL) that aims to simplify the creation of simulations. At a higher level of abstraction, users can design experiments (network topology, neuron and synapse properties, observation units...) without having to deal with core features such as synchronization or actor policies (see Appendix A.2). The DSL also allows the definition of different stages for the simulation (e.g., splitting the simulation into a training phase and a test phase).

Table 3.1 provides a feature-wise comparison of N2S3 with other simulators. Table 3.2 shows that N2S3 offers a reasonable efficiency: when the neural activity is sparse enough, it can run quicker than clock-based simulators thanks to its event-based paradigm. N2S3 is freely available at `https://sourcesup.renater.fr/wiki/n2s3` under the CECILL-B licence.

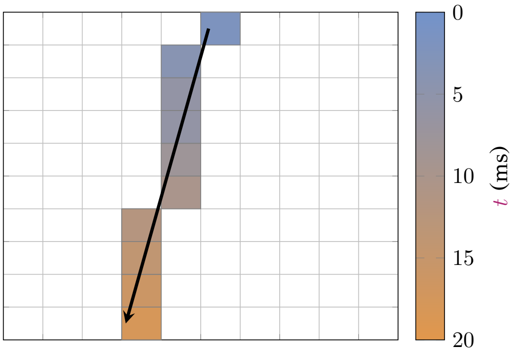


Figure 3.2: Example of an input from the motion detection task. The arrow shows the theoretical orientation and direction of the motion and the squares depict the pixel activations of this input.

3.1.1 Case study: motion detection

The flexibility of N2S3 is demonstrated by using three different designs of a neural network to solve a motion detection task. An additional constraint of using small networks is added, so that these networks can be more easily implemented on hardware. The task consists in detecting the direction of the motion of a pixel on a two-dimensional grid. Each benchmark consists in a series of successive movements of a pixel on the grid, without any overlap between two inputs. Each motion has a linear trajectory, a constant velocity, and a direction, and so, is represented by the successive activation of the different pixels of the trajectory.

Two datasets are used in this section, a basic one and a more complex one. For both datasets, the grid dimension is set to 10×10 to maintain a reasonable size for the networks. The simple dataset includes only four directions (up, down, left, right), no variability of the orientations of the trajectories (i.e. each input has an orientation which has an angle to the x-axis of 0, $\frac{\pi}{2}$, π, or $\frac{3\pi}{2}$) and only one possible velocity (0.5 pixels/ms). In order to avoid that all the trajectories pass through the center of the grid, a pixel is chosen at random as the reference point of the trajectory for each sample. Each trajectory is parallel to one axes of the grid, and so all samples contain exactly 10 successive stimuli. The complex dataset has eight possible directions (the diagonals are added), and a velocity range from 0.25 to 0.5 pixels/ms. To introduce some variability, a Gaussian noise is added to the orientation of each trajectory. Furthermore, a random orthogonal shift is applied to each trajectory; it follows a normal distribution $\mathcal{G}$ with the grid center as its mean. Finally, some jitter noise can be applied to the spike timestamps to observe the tolerance of the network to temporal variations.

The classification score of the task is computed by taking the ratio between the number of well-classified motion samples and the total number of samples. A motion is considered as well classified if the first output neuron to fire since the beginning of the motion corresponds to the class of the current motion. Neurons are associated to the class on which they react the most for a set of input sample.

Thanks to the flexibility of N2S3, each approach can be tested and compared to the others. An interesting property of motion detection is that not only synaptic weight learning is important: synaptic delays play an essential part in the resolution task. According to the incoming temporal pattern, delays allow synchronizing the post-synaptic spikes. Thus, neurons will fire only when some specific input patterns arise [173]. Therefore, setting and learning the synaptic delays are also a required feature. Several topologies and several learning processes are used in the different networks. Three approaches are retained: reservoir computing with both weight and delay learning, a small feed-forward network with a global unsupervised

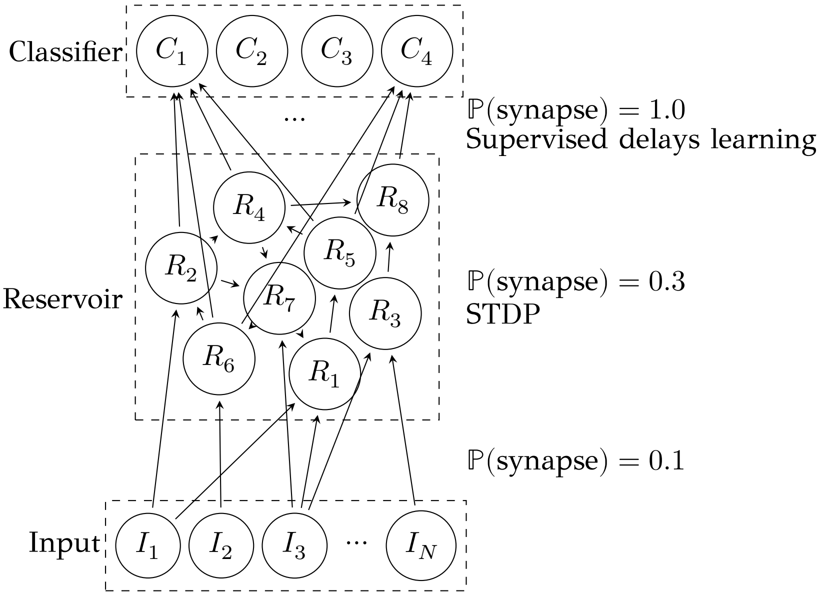


Figure 3.3: Topology used in the reservoir computing approach.

training of delays coupled to a supervised training of weights, and a FF network without training (i.e. all weights and delays are manually set to solve the task).

3.1.2 Comparison of the Three Approaches

Reservoir Computing Approach

Our first approach to solve this motion detection task is inspired by [174], in which the authors use reservoir computing (RC) and supervised delay adaptation to learn two different patterns. The basic principle of RC is to couple a recurrent, randomly connected, layer (called the reservoir) with a linear classifier (called the readout). Only the classifier needs to be trained in order to map the state of the reservoir to a class (see Figure 3.3). More details about the network topology and the training algorithm are available in [174]. A first study consists in evaluating the influence of the reservoir size on the classification score. With a larger reservoir, the network yields a higher classification score: this makes sense since a larger number of neurons means more states to classify the current input (see Figure 3.6). A second study is about the impact of STDP on the state of the reservoir. STDP slightly improves the classification rate when it is activated within the reservoir. STDP will improve the recognition of repeated patterns inside the reservoir, which helps the readout to improve classification performances.

Trained Feed-forward Approach

The second approach is an application-specific FF network. This approach aims to create the smallest topology possible that can perform well on this task. This topology is a compound of three layers. The first layer reduces the dimensionality of the inputs. The two orientations, vertical and horizontal, are each mapped to a different sub-network. The second layer recognizes the input velocity, in every orientation. Experiments show that ten velocity classifiers per trajectory (five per direction) are enough. Finally, the third layer classifies the directions by adjusting its synaptic weights (see Figure 3.4). An unsupervised method is used to adapt the synapse delays in the second layer:

$$\Delta_{\mathrm{d}} = \eta(t_{\mathrm{post}} - t_{\mathrm{pre}} - d) \tag{3.1}$$

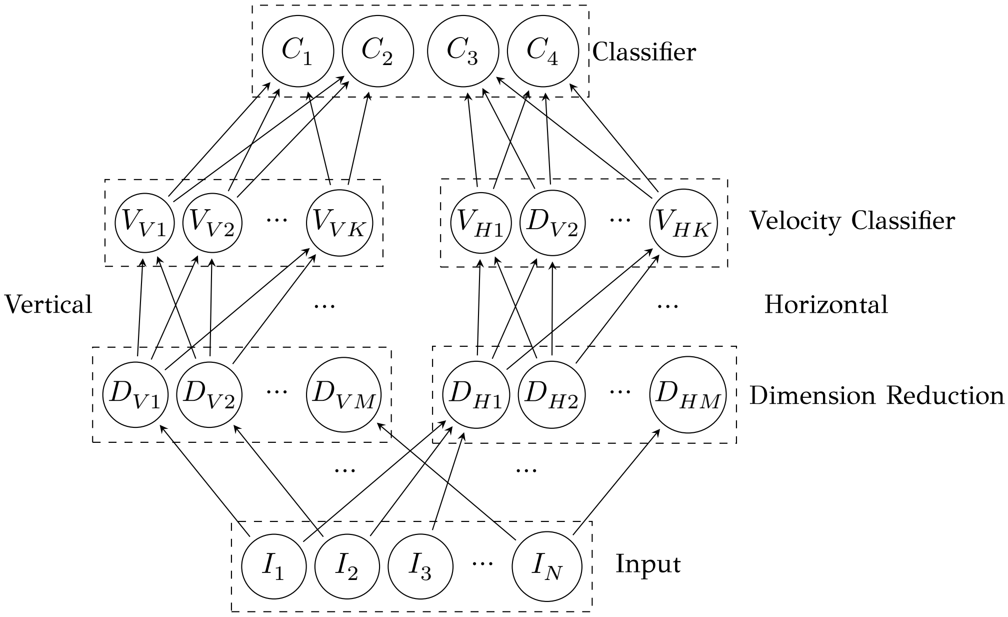


Figure 3.4: Topology used in the trained feed-forward approach.

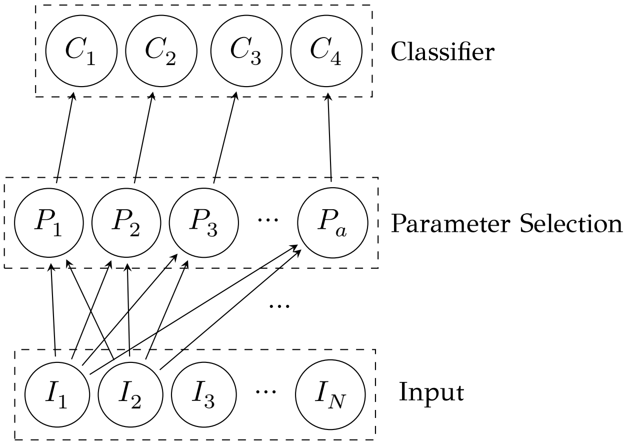


Figure 3.5: Topology used in the fixed feed-forward approach.

with Δ_d the variation of the delay, η the learning rate, d the current synaptic delay, t_pre and t_post the fire timing of input and output neurons.

A supervised STDP rule is used in the third layer to map each velocity to the correct direction. LTP is applied on the i$^\mathrm{th}$ output neuron if the current pattern belongs to the i$^\mathrm{th}$ class:

$$\Delta_\mathrm{w} = \begin{cases} \eta_\mathrm{w} e^{-\frac{t_\mathrm{pre}-t_\mathrm{post}}{\tau_\mathrm{STDP}}} & \text{for } n_i \text{ if the input is } c_i \\ -\eta_\mathrm{w} e^{-\frac{t_\mathrm{post}-t_\mathrm{pre}}{\tau_\mathrm{STDP}}} & \text{otherwise} \end{cases} \tag{3.2}$$

Such an application-specific network has the advantage of achieving a better score with only 44 neurons and 480 synapses (see Figure 3.6), but has to be designed specifically for a given task.

Fixed Feed-forward Approach

The third approach is a fixed network. All the synaptic weights and delays are fixed at the creation of the network, and so, the network is not trained at all. Since

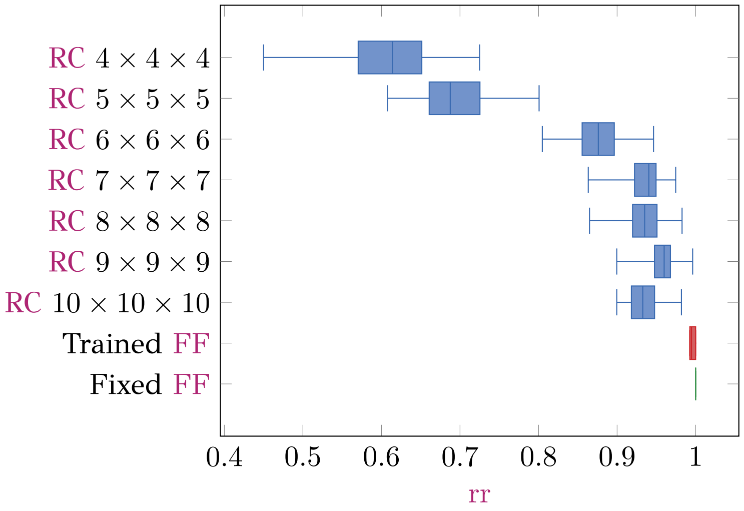


Figure 3.6: Results of classification with the different approaches. Each configuration is run 100 times. For reservoir computing, the size of the network is indicated (e.g. $4 \times 4 \times 4$ mean that the reservoir consists of 64 neurons). The network topology is regenerated randomly in each run.

the training algorithms of SNNs are not well mastered yet, manually setting the parameters can have the advantage to produce a network better suited to solve the task, as long as the task is simple enough for a human to find optimum parameters.

The network consists of two FF layers. The first layer aims to cover a maximum of possible cases that can arise from the input data. The neurons of the first layer cover each a specific case. After testing different configurations, three parameters are retained to define these cases: the orientation of the trajectory m_Θ, the orthogonal shift m_S, and the velocity m_V. The incoming synaptic delays and weights are defined by the following equations:

$$d_{i,j} = ||\overrightarrow{x_{i,j}m_\text{S}}|| \times \cos\left(\widehat{x_{i,j}m_\Theta}\right) \times m_\text{V}$$

$$w_{i,j} = \begin{cases} 1 & \text{when } \min(\text{pd}(x_{i,j}, m_{\min\Theta}), \text{pd}(x_{i,j}, m_{\max\Theta})) < \text{pd}_\text{max} \\ 0 & \text{otherwise} \end{cases}$$

where i and j are the synapse coordinates on the input grid, $x_{i,j}$ the point at coordinates (i, j), m_S a reference point of the trajectory, m_V the current velocity, m_V the current orientation, $m_{\min\Theta}$ and $m_{\max\Theta}$ the bounds of the orientation, and pd the perpendicular distance from a point to a line. The second layer aims to map each selection neuron to the associated class. According to the orientation parameter of each neuron of the first layer, a unique outgoing connection will be created to the classifier neuron associated to the orientation class (see Figure 3.5). This approach provides excellent results (see Figure 3.6), but requires a large number of neurons and synapses to cover enough parameters (208 neurons and 20,200 synapses).

3.1.3 Energy Consumption

In many types of applications, using SNNs can help to save a large amount of energy as compared to the same applications on classic von Neumann architectures. However, since a comparison with such architectures is very difficult to realize (estimating the energy consumption of a program instruction is a difficult issue because of the numerous complex hardware and software mechanisms involved), only the different SNN approaches are compared to each other.

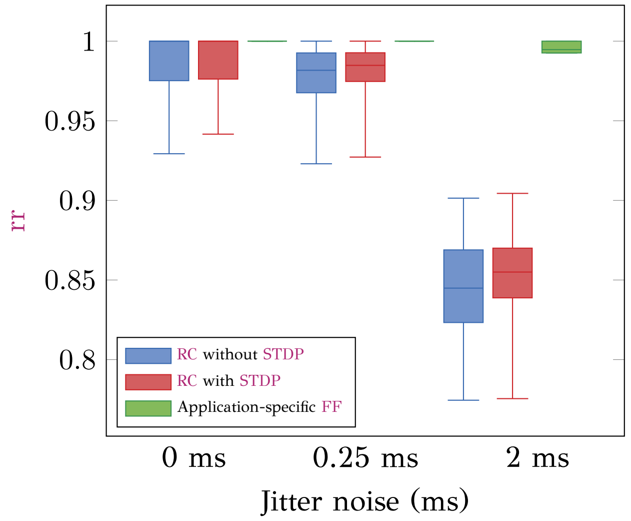


Figure 3.7: Comparison of recognition rates of several networks under different levels of jitter noise. RC networks use a reservoir of size $7 \times 7 \times 7$. Each configuration is run 100 times. STDP improves the recognition rate. As expected, the FF network performs well against the reservoir computing network, while having fewer neurons and synapses (44 neurons and 880 synapses vs. 347 neurons and approximately 20,000 synapses).

Table 3.3: Estimation of the basic values of energy consumption for the used hardware model [175].

Parameter	Value
e_{fire}	4 fJ
e_{spike}	4 fJ
p_{neuron}	100 pW
p_{synapse}	100 pW

The estimation of the energy consumption of our models can be computed by the following equations:

$$\begin{aligned} e_{\text{dynamic}} &= |\mathcal{D}| \times e_{\text{fire}} + |\mathcal{E}| \times e_{\text{spike}} \\ e_{\text{static}} &= \Delta_t \times (p_{\text{neuron}} \times |\mathcal{N}| + p_{\text{synapse}} \times |\mathcal{S}|) \\ e_{\text{total}} &= e_{\text{dynamic}} + e_{\text{static}} \end{aligned} \tag{3.3}$$

with $\mathcal{D}$ the set of neuron firing events, e_{fire} the energy consumed when a neuron fires, $\mathcal{E}$ the set of spikes passing through the synapses, e_{spike} the energy needed to transmit a spike through a synapse, Δ_t the duration of the measurement, $\mathcal{N}$ the set of neurons in the network, p_{neuron} the power dissipated by one neuron, $\mathcal{S}$ the set of synapses in the network, and p_{synapse} the power dissipated by one synapse.

Table 3.3 lists the properties of our hardware model and Table 3.4 shows the simulated energy consumption of the three architectures considered. While conventional artificial neurons exhibit an energy efficiency in the range of 1 pJ/spike, it is worth noting that the dynamic power is far much lower than the static power in our case [175]: as described by Table 3.4, the energy efficiency of the implemented model (4 fJ per spike) is negligible as compared to its static power (100 pW).

Each approach has its own benefits, when comparing its energy consumption and its performances. On the one hand, reservoir computing is the most general approach. However, in order to obtain satisfactory results, it is necessary to use a large reservoir, which rapidly increases the amount of neurons and synapses and so,

Table 3.4: Estimation of the energy consumption of the different approaches. The results averaged over 100 runs. Our estimations show that the consumed dynamic energy is negligible.

Network	$e_{dynamic}$ (mJ)	e_{static} (mJ)	e_{total} (mJ)	$\lvert\mathcal{N}\rvert$	$\lvert\mathcal{S}\rvert$
RC $4 \cdot 4 \cdot 4$	$2.6e^{-4}$	8.5	8.5	68	~1,631
RC $7 \cdot 7 \cdot 7$	$1.3e^{-3}$	100.30	100.31	347	~19,714
RC $10 \cdot 10 \cdot 10$	$4.0e^{-3}$	526.24	526.25	1,004	~104,245
Trained FF	$2.3e^{-4}$	2.62	2.62	44	880
Fixed FF	$1.3e^{-3}$	102.04	102.04	208	20,200

the energy consumption. On the other hand, using a fixed network yields excellent results on the reference dataset, but again, a large network is required to have a good coverage of the parameters. Thus, the trained feed-forward network is a good candidate because it provides a very good compromise between the network size, and so the energy consumption, and the task performance. Even if this trained feed-forward network remains task-specific, it can be retrained on a different dataset.

3.1.4 Conclusion

N2S3 is a hardware spiking neural network simulator design to be scalable, flexible and easy to use. The study of a motion detection case allows to demonstrate the flexibility of this simulator. This flexibility concerns the network topologies (feed-forward or recurrent) and the learning approaches (local or global, supervised or unsupervised, weight or delay learning). N2S3 can also evaluate both the generalization performance and the energy consumption of these various networks built with a very low power complementary metal-oxide semiconductor (CMOS) design.

3.2 CSNNS

Since N2S3 is a general purpose and flexible simulator, many optimizations cannot be used. For example, N2S3 creates an object instance for each neuron and synapse, in order to allow using different models in the network. In return, such practice tends to use much memory, and prevent the usage of SIMD instructions. As a consequence, a specific simulator for the models defined in the following chapters of this manuscript has been developed, named CSNNS, in order to improve the simulation time and memory usage of these SNNs. For example, spike timestamps can be stored in matrix and processed parallelly since the simulator only supports at most one spike per neuron per sample (i.e. as in temporal coding). Currently, CSNNS is optimized to run on CPUs, by using SIMD instructions, but other backends, such as GPUs, could be added in the future. This simulator is written in C++ with the Qt library to manage the plots. In order to facilitate the follow-up of experiments, a file is automatically created with all the parameters of the models at the beginning of a simulation (see Appendix A.3). Moreover, it is possible to load these files to recreate the configuration of a previous experiment, and, so, reproduce it. This simulator is also event-driven, but unlike N2S3, basic building blocks are not neurons or synapses, but layers. This allows using SIMD instructions to speed up the simulation. CSNNS is used in Chapter 5 and Chapter 6. Table 3.5 shows the comparison in term of memory and execution duration between N2S3 and CSNNS, which clearly demonstrate the benefits of using CSNNS for simulating these specific models.

Table 3.5: Comparison of simulator performances on the same configuration (Mint 19.1, i7 core, 32GB RAM). For each simulator, a convolution column of 64 filters of size 5×5 is trained for 1 epoch (60,000 patches).

Measure	N2S3	CSNNS
Execution duration	0:14:36	0:00:08
Memory usage	349 MB	741 MB

Even if this simulator is dedicated to a limited range of models, it was designed in such a way that it can be optimized for specific cases, thanks to the simulation policy. In the case of layer-wise learning (i.e. training one layer at a time, from the input to the output of the network), one of the most straightforward policies allows using little memory. Each sample is recomputed from the input until the current layer to train. This policy can be computationally expensive since it requires to apply the different pre-processing methods, and then simulate all the layers from the input to the currently trained layer. A second policy allows to speed up the simulation, but requires much more memory: instead of recomputing each time the spike trains generated at the different layers, intermediate representations are saved in memory. So, only one layer is simulated at each step since the input of this layer is already available. Other optimizations are also available, such as using dense or sparse tensors to save memory if the current model allows it.

An experiment with this simulator can be defined in the following way. First an experiment object with a simulation policy should be created. Some pre-processing operations can be added, such as on/off filtering (see Section 2.3.1), value scaling, or pooling. These pre-processing operations are responsible for transforming each input sample into another tensor. After the pre-processing definition, an input converter should be defined, in order to transform the input tensor into spikes. The next step is to add the training set and test set to the experiment. The simulator offers the basic support of the MNIST, CIFAR (10 and 100), Caltech (101 and 265), and STL datasets. The support of other datasets can be easily added. The next part of the experiment consists in defining the SNN architecture, by successively defining the layers and their parameters, and the number of epochs required to train each layer. Optionally, a list of visualization methods can be specified, such as the histogram of the distribution of spike timings, the reconstruction of the receptive field of neurons, or the evolution of the thresholds. Finally, the last part of the experiment consists in a list of outputs. Each output consists in an output converter (i.e. to transform back spikes into tensors), a list of post-processing steps, to apply operations on tensors , and a list of evaluation metrics. For example, it is possible to get sparsity and coherence metrics, but also get the recognition rate obtained after classification by an SVM. A complete example of an experiment is given in Appendix A.3.

Like N2S3, CSNNS is under the CECILL-B licence and so, freely available, at https://gitlab.univ-lille.fr/bioinsp/falez-csnn-simulator.

3.3 Conclusion

As discussed in the beginning of this chapter, the study of SNNs answers to multiple challenges, which each have their own specificities. Different simulation tools are thus required to adapt to the different requirements (see Section 2.4). N2S3 aims to simulate the behavior of hardware, and thus, focus on such models. This simulator is intended be flexible, scalable, and easy to use. This flexibility comes to the cost of some overhead that has an impact on simulation speed and memory usage. N2S3 is designed to be scalable, thanks to its actor-oriented paradigm. However, some work is still needed to make the distribution of simulations really efficient. Finally, thanks to the DSL feature of Scala, N2S3 offers a friendlier interface to design experiments. This DSL interface needs to be extended in order to cover all the features offered by N2S3.

Unlike N2S3, CSNNS is designed to simulate a limited number of models. In return, this constraint allows to widely optimize the simulation speed or the memory usage (see Table 3.5). This simulator allows to run large networks in a short time. Moreover, this simulator helps users archive the experiments metrics according to the used configurations by generating logging files, and by allowing to reload previous experiments.

All the experiments described in the following chapters are simulated either with N2S3 (Chapter 4) or with CSNNS (Chapter 5, Chapter 6).

Chapter 4

Frequency Loss Problem in SNNs

Introducing multi-layered SNNs seems to be a promising way to reach state-of-the-art results on computer vision datasets. It is acknowledged that deep hierarchical representations improve the expressiveness of models [46], and yield state-of-the-art performance on many tasks [47], [48]. However, most SNNs reported in the literature are single-layered [176]–[178]. However, using multiple layers is necessary to perform complex tasks [46]. Although multi-layer SNNs exist, their performances in many tasks are far behind deep ANN [151], [179], or they rely on non-spiking mechanisms [7], [150], which limits their benefits. Maintaining a sufficient spiking activity throughout the layers is crucial, because spikes are used to transmit information and are also necessary for learning. However, this constraint is often set aside, as authors rather focus on recognition rates [111], [112]. To illustrate this, a two-layer SNN is trained (see Table 4.1) using the parameters of [111]. When using WTA inhibition, the spiking activity becomes null after only two layers. Even after releasing the inhibition constraint, the output frequency remains much lower than the input frequency, which does not allow any training in subsequent layers.

New mechanisms are proposed in Section 4.1 which allow maintaining a desired frequency, while keeping a good classification rate.

- Target frequency threshold adaptation (Section 4.1.1): a method to adapt the threshold of neurons in order to reach a desired output spiking frequency, using an online unsupervised learning rule.

- Binary coding (Section 4.1.2): a process to generate input spike trains that maintains the output spiking frequency within layers and facilitates the setting of the model parameters.

- Mirrored STDP (Section 4.1.3): a modification of STDP rules that takes advantage of binary coding to improve the learning speed and to reach a more stable network state.

Section 4.2 presents the experiments that validate these mechanisms in the case of single layer SNNs.

Table 4.1: Average frequencies of a two-layer SNN using LIF neurons (see Section 2.2.1), multiplicative STDP synapses (see Section 2.3.3), LAT (see Section 2.2.6), and frequency coding (see Section 2.2.3).

Methods	Input Frequency	Layer 1 Frequency	Layer 2 Frequency
Winner-Take-All	2.0396	0.0484	0.0000
Soft Inhibition	2.0407	0.2362	0.0106

4.1 Mastering the Frequency

4.1.1 Target Frequency Threshold

We introduce a threshold adaptation mechanism that provides better control over the output frequency: target frequency threshold (TFT) adaptation. In contrast to LAT, this method allows the explicit specification of the target frequency. First, the objective output frequency F_{expected} that the neuron should reach is defined, depending on the neural coding used. When using frequency coding, F_{expected} can be computed as:

$$F_{\text{expected}} = \rho \times \frac{t_{\text{exposition}}}{t_{\text{exposition}} + t_{\text{pause}}} \times F_{\text{max}} \tag{4.1}$$

with ρ the expected sparsity output factor, which is $\frac{1}{|l_{\text{output}}|}$ (i.e. $|l_{\text{output}}|$ is the number of neurons in the output layer) in the case of WTA inhibition because only one neuron can discharge at any given time step. $T_{\text{exposition}}$ and T_{pause} are respectively the presentation duration for one sample and the duration of the resting period between two samples. F_{max} is the frequency that represents the largest input value. Then, the actual frequency F_{actual} can be computed with the following formula:

$$F_{\text{actual}}(t + t_{\text{update}}) = \gamma \times F_{\text{actual}}(t) + (1 - \gamma) \times \frac{|\mathcal{E}|}{t_{\text{update}}} \tag{4.2}$$

with γ the update factor ($\gamma = 0.9$ is fixed in all the experiments), t_{update} the duration of the update window ($t_{\text{update}} = t_{\text{exposition}} + t_{\text{pause}}$ in this chapter) and $\mathcal{E}$ the set of output spikes emitted by a neuron during t_{update}. $\frac{|\mathcal{E}|}{t_{\text{pause}}}$ gives the current frequency, added to the previous frequency with weight $1 - \gamma$. From these, the threshold can be periodically updated to rectify the difference between the actual frequency and the objective frequency:

$$v_{\text{th}}(t + t_{\text{update}}) = v_{\text{th}}(t) + \eta_{\text{th}} * (F_{\text{actual}}(t + t_{\text{update}}) - F_{\text{expected}}) \tag{4.3}$$

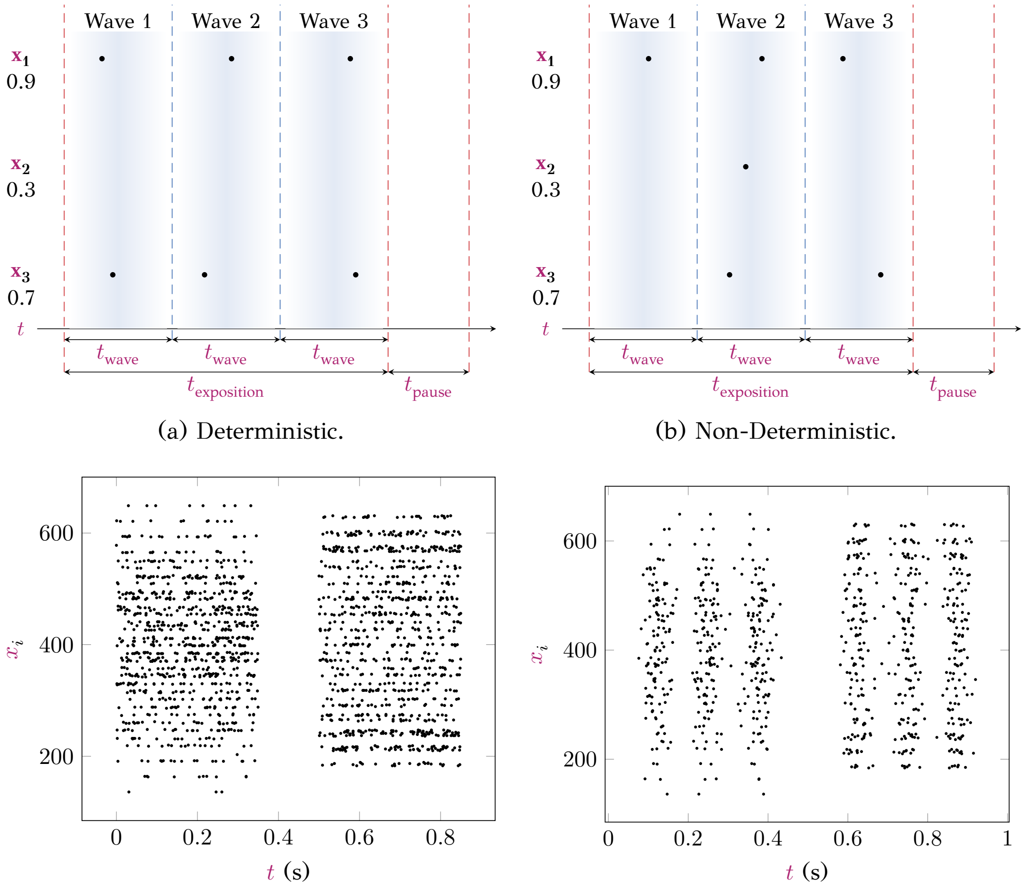
with η_{th} the threshold learning rate (η_{th} is set to 0.1).

Working at higher frequencies means decreasing integration periods. It decreases the amount of information carried by the frequency since fewer spikes are available to trigger a neuron discharge. This may result in a misrepresentation of the input patterns, and lead to lower recognition rates. For this reason, a neural coding is introduced that provides a more effective pattern representation at high frequencies.

4.1.2 Binary Coding

In order to improve the synchronization of output spikes, spike trains are generated as a cycle: a positive phase with spikes and a negative phase without spikes. It produces ”spike waves” (see Fig. 4.1d). Thus, instead of using the frequencies or the timings of spikes to code input values, the only presence or absence of spikes in a wave is used to represent the input sample. The timing of a spike in a wave is assumed to carry no information in this coding, and so, is meaningless. It makes the coding more flexible: it is less sensitive to variations in time constants of the model. Neurons use only the spikes of a single wave to fire, which facilitate the maintenance of the frequency. Two strategies to determine whether a wave contains a spike are used:

- deterministic binary coding (see Fig. 4.1a): it uses a fixed threshold x_{th} on the input values. If the value is above x_{th}, the wave contains one spike, and none otherwise;



(a) Deterministic.

(b) Non-Deterministic.

(c) Frequency coding example.

(d) Binary coding example.

Figure 4.1: Generation of a spike train in binary coding by the deterministic (a) and the non-deterministic (b) strategies, and comparison of spike trains generated by frequency coding (c) and binary coding (d) from two input samples.

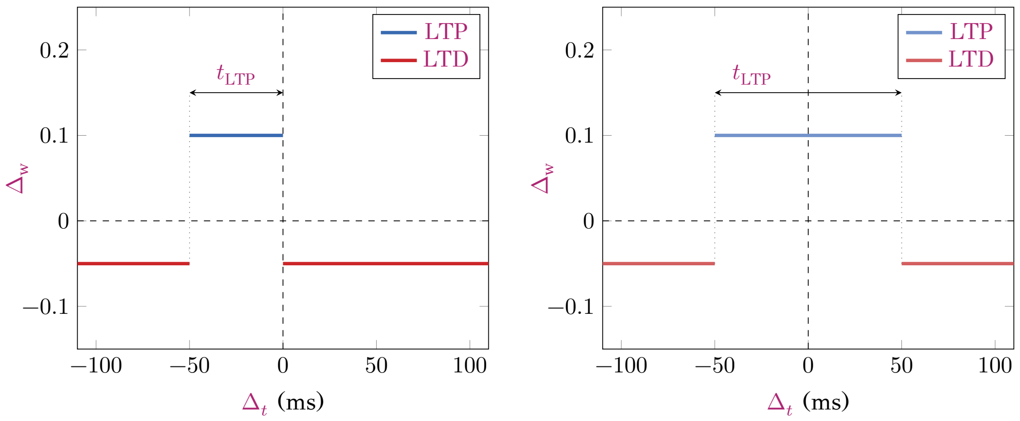
- non-deterministic binary coding (see Fig. 4.1b): the input value is seen as the probability that the wave contains a spike, independently of the other waves and inputs.

WTA inhibition requires that spike timings are not simultaneous. To respect this constraint, the waves are produced by generating spike timings following a normal distribution, $t \sim \mathcal{G}(t_{\text{wave}}, \sigma_{\text{wave}})$, where t_{wave} is the mean of spike timing of the wave and σ_{wave} the variance of spike timings. Thus, the wave f_{in} for the input value x is generated by the following equation:

$$f_{\text{in}}(x) = \begin{cases} \{\mathcal{G}(t_{\text{wave}}, \sigma_{\text{wave}})\} & \text{if } \text{cond}(x) \\ \emptyset & \text{otherwise} \end{cases} \tag{4.4}$$

with $\text{cond}(x)$ the condition used to determine the presence of a spike in a wave according to the strategy used.

Such coding is suited to the multiplicative STDP rule (Equation 2.28), because the exact timing of a spike has no influence in this rule. Instead, multiplicative STDP reinforces all the connections where an input spike is present in the LTP window, which can be fixed to match the wave duration t_{wave}.the learning process only checks for the presence of a spike in the wave, ignoring its actual timing.. To enhance performance, the wave of an input sample can be replicated. This reduces the impact of the randomness of the generation process. The first waves can be



(a) Multiplicative STDP.

(b) Mirrored STDP.

Figure 4.2: Difference between the multiplicative STDP rule (Equation 2.28) (a) and the mirrored STDP rule (Equation 4.6).

used to perform an early classification, and the following waves to improve this first estimate. The non-deterministic process behaves like a binomial distribution, so a sufficient number of waves can provide an accurate estimate of input values. With this coding, the expected output frequency can be estimated as follows:

$$F_{\text{expected}} = \rho \times \frac{n_{\text{wave}}}{n_{\text{wave}} \times t_{\text{wave}} + t_{\text{pause}}} \tag{4.5}$$

This coding scheme makes it easier to adjust the parameters of the model, using the following dependency constraints:

- τ_{leak} should be wide enough to maintain the potential along the wave ($\tau_{\text{leak}} = t_{\text{wave}}$);
- t_{pause} should be long enough to let potentials go down before the next input. For the LIF model, $t_{\text{pause}} = 4\tau_{\text{leak}}$. Following the analytic form of the model, $v(t + t_{\text{pause}}) = v(t) \times \exp(-\frac{4\tau_{\text{leak}}}{\tau_{\text{leak}}}) = 0.02v(t)$, i.e. v is decreased by 98%;
- σ_{wave} should be large enough to allow the propagation of inhibition spikes.

4.1.3 Mirrored STDP

The way in which timestamps are generated with binary coding requires to take into account not only pre-synaptic spikes that occur before post-synaptic spikes, but all the spikes of the current input wave: pre-synaptic spikes that arrive shortly after the post-synaptic spike should contribute to the pattern. Based on this statement, the multiplicative STDP rule (Equation 2.28) was extended by centering the LTP window on t_{post} (see Fig. 4.2b). With this new rule, weights increase in the presence of a spike so that $\left|t_{\text{pre}} - t_{\text{post}}\right| < \frac{t_{\text{LTP}}}{2}$:

$$\Delta_w = \begin{cases} \eta_{w+} e^{-\beta \frac{w - w_{\min}}{w_{\max} - w_{\min}}} & \left|t_{\text{pre}} - t_{\text{post}}\right| < \frac{t_{\text{LTP}}}{2} \\ -\eta_{w-} e^{-\beta \frac{w_{\max} - w}{w_{\max} - w_{\min}}} & \text{otherwise} \end{cases} \tag{4.6}$$

Combined to binary coding, t_{LTP} can be set to cover most of the wave duration. For instance, $t_{\text{LTP}} = 4\sigma_{\text{wave}}$ ensures that over 99.99% of spikes are contained within the LTP window. This type of learning rule, also called symmetric STDP, was observed *in vivo* alongside the original STDP [180].

Table 4.2: Parameters used during the experiments.

LIF Neuron					
$v_{\text{th}}(0)$	10 mv	τ_{leak}	100 ms	r_{m}	1 Ω
t_{ref}	10 ms	v_{rest}	0 mV		
Multiplicative STDP Synapse					
$\eta_{\text{w}+}$	0.005	$\eta_{\text{w}-}$	0.01	β	2.0
w_{min}	0.0	w_{max}	1.0		
Frequency Coding					
$t_{\text{exposition}}$	350 ms	t_{pause}	150 ms	F_{max}	22 Hz
Binary Coding					
t_{wave}	100 ms	t_{pause}	400 ms	σ_{wave}	5 ms
x_{th}	0.5				
Leaky Adaptive Threshold					
Θ_{+}	0.05 mv	Θ_{leak}	10,000 seconds		
Target Frequency Threshold					
γ	0.9	η_{th}	0.1		

4.2 Experiments

4.2.1 Experimental Protocol

All experiments consist in training one fully connected SNN layer over one epoch of the MNIST dataset [65]. This network consists of LIF neurons [112] and WTA inhibition [111] with no delay, which is implemented by lateral inhibition connections. The neural coding used is either frequency coding (Section 2.2.3) or binary coding (Section 4.1.2). Classification is performed by assigning to each output neuron the class for which it is most active, as in [112]. The network has 784 inputs (corresponding to the pixels of 28×28 images), and a variable number of output neurons $|l_{\text{output}}|$: 16, 32, 64, 128, 256, and 512. The parameters used in the experiments are given in Table 4.2. All results are averaged over 10 runs. The experiments are implemented using the N2S3 simulator [167].

The binarization threshold x_{th} of deterministic binary coding is set to 0.5. This value has little impact on the MNIST dataset since most pixel values are close to 0 or 1 (Fig. 4.3).

4.2.2 Target Frequency Threshold

Our first study compares the proposed TFT (Section 4.1.1) to LAT (Section 2.2.6). Table 4.3 shows the recognition rates and the output frequencies of frequency coding using both LAT and TFT. Up to 128 output neurons, LAT reduces the output frequency. With 16 neurons, LAT results in a relative difference of 58.96% between F_{actual} and F_{expected}. With TFT, F_{actual} is very close to the objective frequency F_{expected} (e.g. a relative difference of −0.79% with 64 neurons). However, with TFT, the recognition rate is not as good as LAT due to the higher output frequency (e.g. 78.54% for TFT against 81.74% for LAT with 64 neurons): as suggested in Section 4.1.1, working with higher output frequencies and frequency coding decreases the number of integrated incoming spikes and, therefore, the quality of the pattern representation. However, TFT with 16 output neurons results

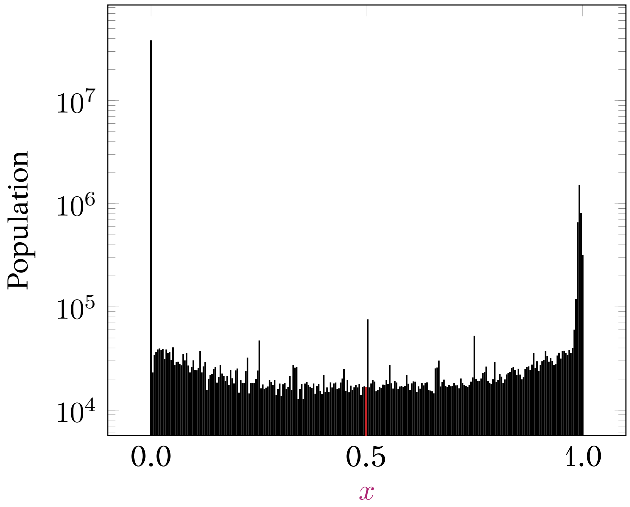


Figure 4.3: Histogram of the values in the MNIST training set in logarithmic scale. Most of the values are close to 0 or 1.

Table 4.3: Recognition rates and frequencies of LAT and TFT for different layer sizes. Δ_F is the relative difference between F_{expected} and F_{actual}.

$\lvert l_{\text{output}}\rvert$	F_{expected}	LAT			TFT		
		rr	F_{actual}	Δ_F	rr	F_{actual}	Δ_F
16	0.9625Hz	61.68%	0.3950Hz	58.96%	66.89%	0.9607Hz	0.19%
32	0.4813Hz	74.00%	0.2796Hz	41.91%	73.83%	0.4814Hz	−0.02%
64	0.2406Hz	81.74%	0.1864Hz	22.53%	78.54%	0.2425Hz	−0.79%
128	0.1203Hz	86.08%	0.1171Hz	2.66%	82.78%	0.1224Hz	−1.75%
256	0.0602Hz	88.20%	0.0698Hz	−15.94%	85.49%	0.0614Hz	−1.99%
512	0.0301Hz	88.90%	0.0399Hz	−32.56%	87.52%	0.0308Hz	−2.32%

in both a better recognition rate and a higher frequency. Frequency loss is less marked when the number of output neurons increases (e.g. 256 and 512 output neurons). This can be explained partially by a higher probability of simultaneous discharges, which leads to multiple winners with our implementation of WTA inhibition.

4.2.3 Binary Coding

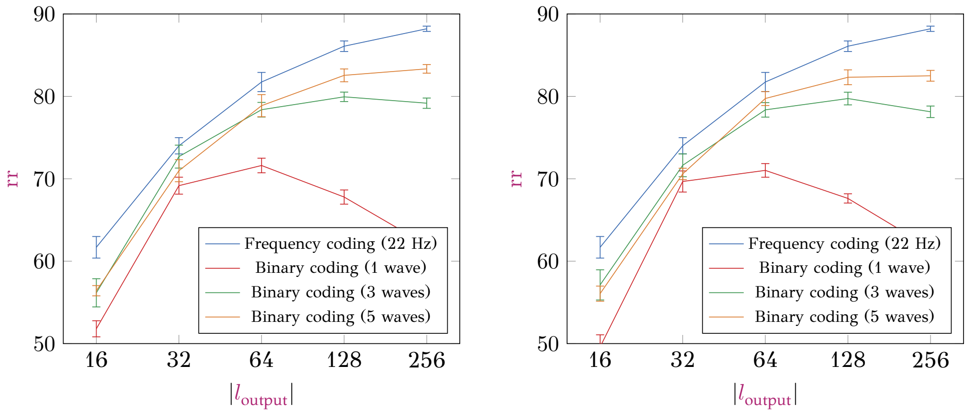
Then, the impact of binary coding on both the recognition rate and the output frequency is investigated. Frequency coding is used as baseline since state-of-the-art models mostly use this coding [112] [111]. Fig. 4.4 and Table 4.4 show the results when using LAT with deterministic and non-deterministic binary coding against frequency coding. Results show that binary coding yields lower recognition rates than frequency coding in the presence of LAT, mostly due to the incorrect output frequencies. Most of the codings result in a difference between F_{expected} and F_{actual} (e.g. $-24,92\%$ of relative difference with deterministic binary coding for $|l_{\text{output}}| = 64$).

Fig. 4.5 and Table 4.5 show the results when using TFT. Binary coding yields better recognition rates than frequency coding in presence of TFT, when using more than one wave.

TFT allows all codings to reach the objective frequency F_{expected}. Moreover, the number of waves impacts the recognition rate. On the first wave, fewer neurons discharge, because some input patterns are too different from the learned patterns.

Table 4.4: Recognition rates and frequencies of the different neural codings combined to LAT, for $|l_{\text{output}}| = 64$.

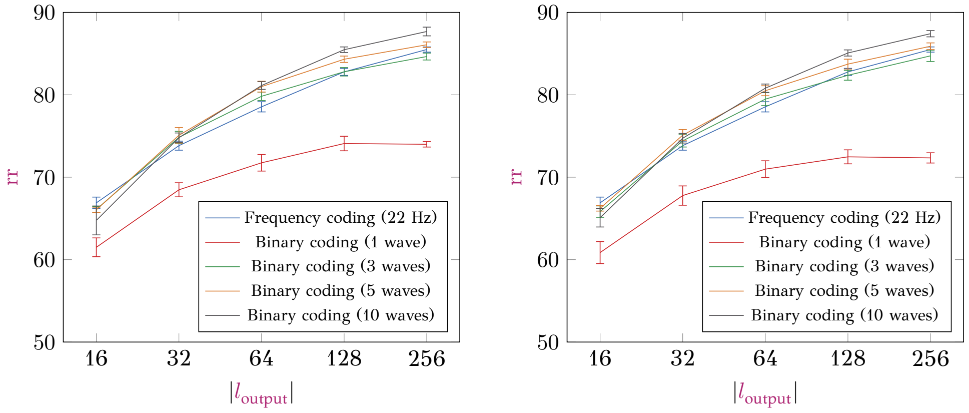
Methods	rr	$\mathbf{F}_{\text{expected}}$	$\mathbf{F}_{\text{actual}}$	$\boldsymbol{\Delta}_{\mathbf{F}}$	$\lvert\mathcal{E}\rvert$
Frequency Coding	81.74%	0.2406Hz	0.1864Hz	22.53%	8.06×10^6
Deterministic (1 wave)	71.61%	0.0313Hz	0.0391Hz	−24,92%	1.06×10^6
Deterministic (3 wave)	78.38%	0.0670Hz	0.0728Hz	−8.66%	3.19×10^6
Deterministic (5 wave)	78.87%	0.0868Hz	0.0889Hz	−2.42%	5.31×10^6
Non-deterministic (1 wave)	71.02%	0.0313Hz	0.0394Hz	−25.87%	1.05×10^6
Non-deterministic (3 wave)	78.36%	0.0670Hz	0.0729Hz	−8.81%	3.15×10^6
Non-deterministic (5 wave)	79.74%	0.0868Hz	0.0885Hz	−1.96%	5.25×10^6



(a) Deterministic

(b) Non-deterministic

Figure 4.4: Recognition rate of the neural coding methods with LAT.



(a) Deterministic

(b) Non-deterministic

Figure 4.5: Recognition rate of the neural coding methods with TFT.

Table 4.5: Performances and frequencies of the different neural coding methods combined to TFT. All configurations use $|l_{\text{output}}| = 64$.

Methods	rr	$\mathbf{F}_{\text{expected}}$	$\mathbf{F}_{\text{actual}}$	$\mathbf{\Delta_F}$	$\lvert\mathcal{E}\rvert$
Frequency Coding	78.54%	0.2406Hz	0.2425	−0.79%	8.08×10^6
Deterministic (1 wave)	71.83%	0.0313Hz	0.0327Hz	−4.47%	1.06×10^6
Deterministic (3 waves)	79.36%	0.0670Hz	0.0692Hz	−3.28%	3.19×10^6
Deterministic (5 waves)	81.06%	0.0868Hz	0.0902Hz	−3.91%	5.31×10^6
Deterministic (10 waves)	81.12%	0.1116Hz	0.1145Hz	−2.53%	1.05×10^6
Non-deterministic (1 wave)	70.98%	0.0313Hz	0.0325Hz	−3.83%	1.05×10^6
Non-deterministic (3 waves)	79.48%	0.0670Hz	0.0693Hz	−3.43%	3.15×10^6
Non-deterministic (5 waves)	80.49%	0.0868Hz	0.0906Hz	−4.37%	5.25×10^6
Non-deterministic (10 waves)	80.80%	0.1116Hz	.1149Hz	−2.88%	1.06×10^6

On average, 7.3% of the test samples do not trigger any discharge on the first wave. Increasing n_{wave} reduces this effect because the remaining potential after the first wave helps neurons reach their threshold. Binary coding also reduces the number of spikes going through the network compared to frequency coding, and provides better recognition rates, both with LAT (Table 4.4) and TFT (Table 4.5), e.g. 8.08×10^6 spikes and 78.54% for frequency coding against 5.31×10^6 spikes and 81.06% for a five waves deterministic binary coding with $|l_{\text{output}}| = 64$. It can lead to more efficient simulations on dedicated hardware as the power consumption depends on the spike dynamics [175]. Also, using a high number of waves yields performance close to frequency coding and LAT: 87.67% for ten waves binary coding against 88.2% when $|l_{\text{output}}| = 256$. So, combining TFT and binary coding provides near-state-of-the-art performance, uses fewer spikes than frequency coding and maintains the objective output frequency. Finally, Fig. 4.6 shows the output distribution of the spike timings. The output distribution is nearly Gaussian, close to the input Gaussian distribution. The mean of the output distribution is higher by a few milliseconds than the mean of the input distribution: the coding introduces only a small latency. This coding preserves the representation of the data over the layers. It makes it possible for subsequent layers to use the same models as the first layer to process data, which is necessary to stack layers. Finally, the number of input spikes triggered after output spikes show the need for mirrored STDP.

4.2.4 Mirrored STDP

Fig. 4.7 shows the performance of mirrored STDP combined with binary coding. Training is faster, i.e. the network can reach a high recognition rate with less training samples than with the multiplicative STDP (Equation 2.28). However, with both STDP rules, the network converges to similar recognition rates after a sufficient amount of training samples. Fig. 4.8 shows that more weights have converged to extreme values (close to 0 or 1) after the training of the network, which means that patterns are more stable during training and neurons are more specialized in recognizing specific patterns.

4.3 Discussion

Binary coding is suited to the MNIST dataset because the values are already nearly binary (as seen in Fig. 4.3), so the binarization step has only little impact on performance. The effect of this pre-processing step on more complex datasets (e.g. CIFAR, ImageNet...) could be questioned.

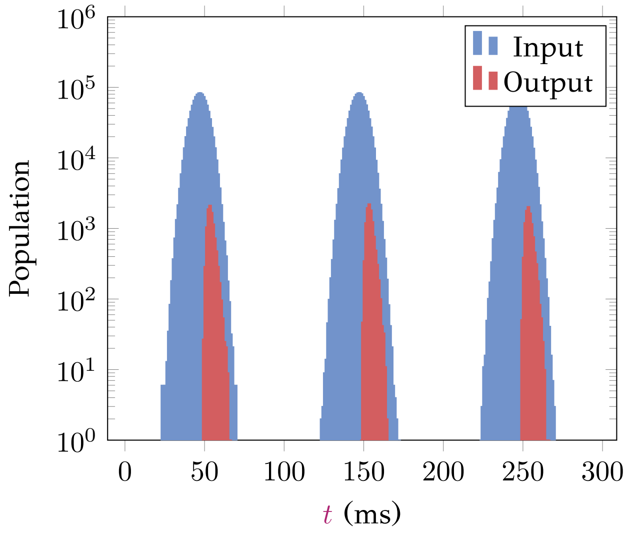


Figure 4.6: Spiking timing distribution at the input and output of the layer for three waves of binary coding. Input timings are Gaussian, and output timings are nearly Gaussian.

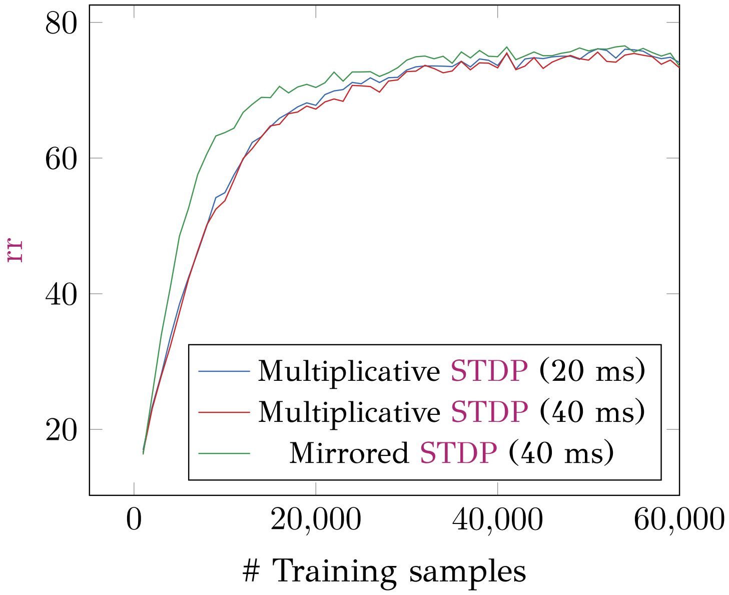


Figure 4.7: Recognition rate on the test set for each STDP rule against training set size ($|l_{\text{output}}| = 64$).

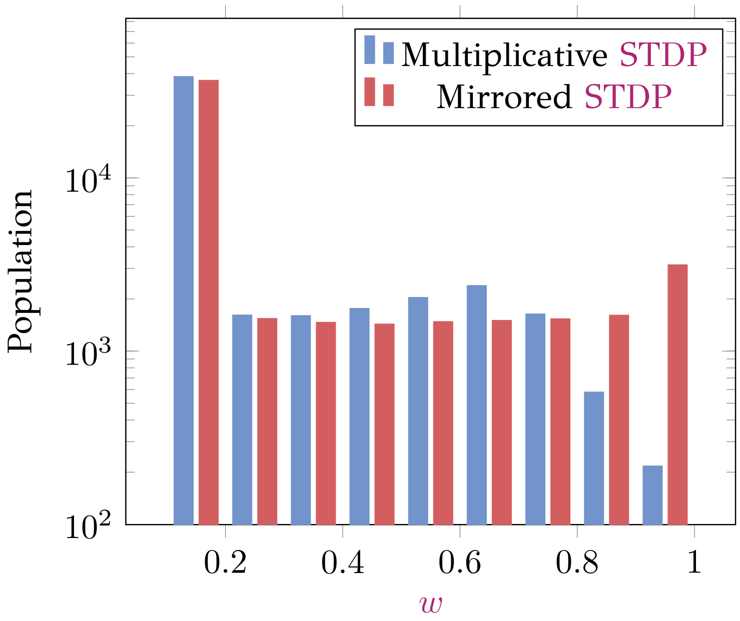


Figure 4.8: Average weight distribution after training (log. scale).

Binary neural networks (BNNs) [181] are a familly of ANNs that use binary activation units and binary weights in the forward part of the network. Such models succeed to reach near-state-of-the-art performances on multiple image classification datasets. However, the training requires to use accurate gradients. Similarly, using STDP on a neural coding that leads to a loss of information, such as binary coding, does not seem to be a good solution to process complex data. Temporal coding seems to be a good alternative candidate as a neural coding since it shares the property of having at most one spike per input per connection with binary coding, but without the loss of information.

This chapter only study models of SNNs constituted by a single layer and with WTA inhibition to ensure that a unique neuron wins at a given time. However, in image recognition, objects are represented by sets of features, corresponding to multiple active units [65] [2]. So, multiple neurons should fire at the same time to have a distributed representation. To do so, further investigation over inhibition mechanisms is necessary.

4.4 Conclusion

SNNs offer an energy-efficient alternative to ANNs, but currently provide poorer performances, notably due to the difficulty of training multi-layer SNNs. They have to maintain a sufficient spiking frequency across their layers, because learning rules such as STDP rely on the presence of spikes. However, standard models from the SNN literature tend to strongly reduce the spiking frequency as a side effect. A three-fold solution is proposed to bypass this problem: TFT adaptation, binary coding, and mirrored STDP. Experiments show that using these mechanisms allows us to maintain both performance and output frequency on a single layer. Next chapters will use these results to set up multi-layered SNNs.

However, the main disadvantage of these mechanisms is the loss of information introduced with binary coding. Even if some work suggests that binary weights and units are sufficient to reach near state-of-the-art-results[142], these models still require accurate floating-point values to propagate errors during training. In the same way, STDP may require accurate representations to perform well. Temporal coding is a good candidate, because it allows to represent continuous values with only one spike, which helps to avoid the frequency loss problem (see Section 4.1.2). However, this coding also introduces more difficulties, such as the low tolerance to jitter noise: an offset of a few milliseconds changes the represented value. In the next chapters, new mechanisms will be proposed in order to learn complex patterns thanks to STDP and temporal coding. These models do not suffer from the frequency loss problem or loss of information due to the coding process.

Chapter 5

Comparison of the Features Learned with STDP and with AE

Currently, most of the SNNs in computer vision are only tested on datasets with limited challenges (rigid objects, limited number of object instances, uncluttered backgrounds...) such as MNIST, 3D-object, ETH-80, or NORB [111], [112], [151], [182], [183], or on two-class datasets [151], [183] (see Section 2.3.3). How they perform on more complex image datasets, what is the performance gap between them and standard approaches, and what needs to be done to bridge this gap is yet to be established. In order to answer to these questions, this chapter proposes mechanisms to train SNNs equipped with STDP on more advanced image recognition datasets (CIFAR-10, CIFAR-100, and STL-10). To do so, the performances of several pre-processing methods are studied in order to improve the usage of STDP on color images. Notably, multiple on/off filtering policies are tested. The whitening transformation is also investigated in Section 5.5.5. Moreover, a novel threshold adaptation mechanism will be introduced in order to be able to learn patterns with temporal coding. This coding avoids the frequency loss problem and the loss of information. Then, a comparison is done with a standard unsupervised feature learning algorithm, sparse AE. From this evaluation, some bottlenecks that need to be addressed are identified in order to push SNNs to the level of standard machine learning approaches for vision applications. As for the previous chapter, only single-layer architectures are evaluated because multi-layer SNN with unsupervised STDP are only very recent and difficult to train, due to the loss of spiking activity across layers (see Chapter 4). This work is one of the first that evaluates features learned by unsupervised STDP-based SNNs on recent benchmarks for object recognition.

5.1 Unsupervised Visual Feature Learning

As seen in Section 2.1.3), minimizing an objective function f_{obj}. However in unsupervised learning, f_{obj} cannot be formulated towards a specific application. Instead, some surrogate objective must be defined, that is expected to produce features that can fit the problem to be solved. Examples include image reconstruction [59], image denoising [60], and maximum likelihood [61]. In some cases, learning rules are defined directly without formulating an explicit objective function, e.g. in k-means clustering [56], but also STDP [109].

In addition to this, additional constraints on the parameters or learning algorithm can be added to regularize the training process and reach better solutions. These constraints reflect assumptions on properties that ”good” visual features should have, such as:

- sparsity: it is often assumed that the extracted features should be sparse, i.e. only a small number of the features can be found in a single image or image region. Sparsity is especially required when the set of features is over-complete[1], to prevent the algorithm from reaching trivial solutions. Sparsity is commonly imposed in sparse coding [184] and auto-encoders [185];

- coherence of features [63]: features should be different to span the space of visual patterns with limited redundancy[2]. Coherence measures the possibility to reconstruct a given feature as a linear combination of a small number of other features, i.e. whether features are locally linearly dependent; coherence should be small, i.e. the dictionary should be incoherent, for the features to be effective.

In Section 5.5, these properties will serve as a basis for the analysis of the tested feature extractor.

5.2 STDP-based Feature Learning

The previous chapter discusses about the advantages and drawbacks of different neural codings. Notably, frequency coding has the disadvantage of needing many spikes to represent accurate values, which can lead to spiking frequency drops. A new neural coding has been introduced in order to bypass this issue, binary coding (see Section 4.1.3). However, the latter has the drawback of losing information, due to the binary nature of the coding. Using multiple repetitions in order to generate a binomial distribution allows improving the performance, but remains an unsatisfactory solution for solving complex tasks. Finally, temporal coding allows representing continuous values with only one spike, which allows avoiding the frequency loss, but also avoiding any loss of information (see Section 2.2.3). In this Chapter and the next, we use latency coding, defined as:

$$f_{\text{in}}(x) = (1.0 - x) \times t_{\text{exposition}} \tag{5.1}$$

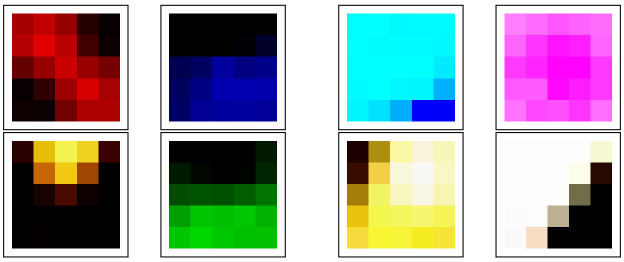
with $x \in [0, 1]$ the input value and $t_{\text{exposition}}$ the duration of the presentation of a data sample. By convention, when $x = 0$, no spikes are generated.

However, this coding is highly sensitive to jitter noise, since an offset of few milliseconds will impact the represented value. Thus, neuron thresholds play a critical role when using latency coding, since the threshold directly impacts the neuron firing timings. A low threshold makes neurons fire early, and so, in latency coding, a high output values is represented. In opposition, high thresholds will make neurons fire late and so, output spikes represent small output values. The firing timing also has an impact on the pattern learned by the neurons. If a neuron tends to fire early, it will have integrated only few spikes, and so, learn almost blank patterns (see Figure 5.1a). Late firing timings let neurons integrate a larger part of the input spikes, and thus, they can learn almost plain patterns (see Figure 5.1b).

In order to facilitate the parameter search step, IF neurons are used because this model uses less parameters compared to other models, such as LIF neurons. Notably, using IF neurons allows to put aside all the issues that can arise from the leak (e.g. balancing the leak so that the beginning of the pattern is remembered, but also being able to forget the previous pattern between two samples). Instead of setting a t_{pause} duration to let the membrane potential v come back to the resting state, a forced reset is performed between each input (i.e. v is set to zero). As in

[1] A dictionary of features is over-complete when its dimension (number of features) is larger than the dimension of the input (size in pixels of the images or image regions processed).

[2] Some authors [186] claim that redundancy should rather be reached to have a good representation, but with limited evidence.



(a) $t_{\text{expected}} = 0.5$

(b) $t_{\text{expected}} = 0.7$

Figure 5.1: Filters learned with different t_{expected}.

the previous chapter, multiplicative STDP is used. In order to add some noise in the timings of spikes, a synaptic propagation delay d is added. In addition to STDP, WTA inhibition ensures that neurons learn different patterns.

New mechanisms are offered in this section in order to allow STDP to learn correctly from color images converted to spikes with latency coding. Section 5.2.1 introduces a threshold adaptation rule, section 5.2.2 offers a method to convert back the spike trains into numerical values and section 5.2.3 presents different pre-processing steps used to learn from color images.

5.2.1 Neuron Threshold Adaptation

The threshold plays two critical roles in SNNs: it highly influences the firing timestamps of the neurons and it allows to maintain the homeostasis of the system. A common method to adapt thresholds in SNNs is to use LAT [112]: when a neuron fires a spike, its threshold is increased to prevent it from firing too often. An exponential leak is applied to help neurons with weak activities. However, this mechanism uses two parameters, which makes the search for suited values difficult (see Section 2.2.6). Moreover, those parameters do not allow to easily converge towards the different types of patterns shown in Figure 5.1. Thus, a new threshold adaptation rule is required to both train the neurons to fire at an objective time t_{expected} and maintain the homeostasis of the network. t_{expected} should be defined within the exposition interval of the input $[0,\ t_{\text{exposition}}]$. Neuron thresholds will be adapted automatically so that the firing timings t converge towards t_{expected} and, at the same time, the homeostasis of the system is maintained.

Each time a neuron fires and each time it receives an inhibitory spike, the threshold is adapted to reduce the difference between the actual time t and the expected time t_{expected}. In this way, all neurons in the competition will apply the same change to their thresholds (i.e. the winner and all the losers), which ensures that the competition is not distorted:

$$\Delta^1_{\text{th}} = -\eta_{\text{th}}(t - t_{\text{expected}}) \tag{5.2}$$

with v_{th} the neuron threshold, t the timestamp at which the neuron fires, and η_{th} the threshold learning rate. This rule corrects the timing error between the actual firing timestamp t and the objective timestamp t_{expected} at each neuron discharge. The optimal value for t_{expected} depends on the dataset; it requires an exhaustive search in the range $[0, t_{\text{exposition}}]$.

Using local and unsupervised learning requires competition mechanisms in order to ensure that neurons learn distinct patterns [111]. WTA inhibition is a straightforward method to do so: only the winning neuron (i.e. the first neuron to spike, since latency coding is used) will apply the learning rule during a pattern and, so, will be able to recognize it. However, the risk of the WTA strategy is that one neuron can take the advantage over the others, and win on every sample (see Section 2.2.6). To guarantee the homeostasis of the system, a second update is applied each time a neuron fires: the winner increases its threshold, while losers decrease their thresholds a little (i.e. when they receive an inhibitory spike):

$$\Delta_{\text{th}}^{2} = \begin{cases} \eta_{\text{th}} & \text{if } t_i = \min\{t_0, \cdots, t_N\} \\ -\frac{\eta_{\text{th}}}{l_{\text{depth}}(n)} & \text{otherwise} \end{cases} \tag{5.3}$$

with l_{depth} the number of neurons in competition in the layer, and t_i the firing timestamp of neuron i. WTA inhibition is used during training: only one neuron is allowed to fire among the $|l_{\text{depth}}|$ neurons on each sample. This mechanism is required to guarantee that neurons will learn different patterns, since only one neuron will apply STDP per sample.

Then, the threshold of the neurons is updated with the following equation:

$$v_{\text{th}}(t) = v_{\text{th}}(t-1) + \Delta_{\text{th}}^{1} + \Delta_{\text{th}}^{2} \tag{5.4}$$

Setting large initial values for the thresholds may prevent the neurons from firing. In the absence of neuronal activity, no learning nor threshold adaptation can be performed. It is therefore preferable to initialize the thresholds with small values to promote neuronal activity within the network.

5.2.2 Output Conversion Function

It is necessary to convert back spike trains into numerical values for the usage of a traditional classifier. Since the spikes are generated with latency coding, an inverse function is needed to create the feature vector $\mathbf{g}$ ($\mathbf{g}_i = f_{\text{out}}(t_i)$):

$$f_{\text{out}}(t) = 1.0 - \frac{t - d_{\text{min}}}{t_{\text{exposition}} + d_{\text{max}} - d_{\text{min}}} \tag{5.5}$$

with f_i the i^{th} output features value, t the spike timestamp (if no spike occurs, then f_{out} return 0), and $[d_{\text{min}}, d_{\text{max}}]$ the range of possible synaptic delay values.

5.2.3 On/Off filters

As STDP learns correlations between input spikes, images are usually pre-processed to help STDP find meaningful correlations. Typically, edges are extracted from the grayscale images, e.g. through a DoG filter [151] (see Section 2.3.1) or Gabor filters [182]. This chapter investigates the application of on/off filtering to color images by offering two strategies. In the first strategy, called RGB color opponent channels, the coding is applied to channels computed as differences of pairs of RGB channels: red-green, green-blue, and blue-red. The second strategy is inspired by biological observations: in the lateral geniculate nucleus, which mainly connects the retina to the visual cortex, three types of color channels exist: the black-white opponent channel (which corresponds to the grayscale image), the red-green opponent channel, and the yellow-blue opponent channel [123]. The second strategy applies on-center/off-center coding to the red-green and yellow-blue (computed as $0.5 \times R + 0.5 \times G - B$) channels. This leads to four possible configurations of image coding: grayscale only, RGB opponent channels, biological color opponent channels (referred to as Bio-color), and the combination of the grayscale channel and the bio-color channels.

5.3 Learning visual features with sparse auto-encoders

AEs [59] are neural networks that perform unsupervised learning by finding latent representations that allow to best reconstruct the input data. In this work, among all variants of AEs and other unsupervised feature learning algorithms, only single-layer AE are considered, for two reasons. First, they belong to the large family of neural networks, as SNNs do, and, within this family, they are one of the most representative models for unsupervised learning (its main competitor being RBMs, which have been shown to optimize a similar criterion [60] and yield comparable performance for visual feature learning [56]). Then, the approach is restricted to single-layer networks, as multi-layer SNNs are only starting to emerge [151]; One-layer SNNs should be well mastered before addressing multi-layer architectures.

The typical architecture of an AE is organized in two parts:

1. An encoder f_{enc}, that maps the input to its latent representation $\mathbf{g}$: $\mathbf{g} = f_{\text{enc}}(\mathbf{X})$.

2. A decoder f_{dec}, that attempts to reconstruct the input from its latent representation: $\widehat{\mathbf{X}} = f_{\text{dec}}(\mathbf{g}) = f_{\text{dec}}(f_{\text{enc}}(\mathbf{X}))$.

The objective function (Eq. 2.7) is thus expressed as:

$$\Phi^* = \arg\min_{\Phi} \; f_{\text{obj}}(\mathbf{X}, \widehat{\mathbf{X}}; \Phi) \tag{5.6}$$

where $f_{\text{obj}}(.,.;\Phi)$ is some measure of the dissimilarity between the input $\mathbf{X}$ and its reconstruction $\widehat{\mathbf{X}}$ given the model parametrized by Φ; in other words, the auto-encoder aims at reconstructing its input with minimal reconstruction error. In the experiments, reconstruction error is measured by the Euclidean distance. The encoder f_{enc} and decoder f_{dec} can be defined as a single-layer or multilayer neural network (in the case of stacked AEs). In the following, only single-layer models of this form are considered:

$$\begin{aligned} \mathbf{g} = f_{\text{enc}}(\mathbf{X}) &= f_\sigma(w_{\text{enc}}\mathbf{X} + \mathbf{b}_{\text{enc}}) \\ f_{\text{dec}}(\mathbf{g}) &= w_{\text{dec}}\mathbf{g} + \mathbf{b}_{\text{dec}} \end{aligned} \tag{5.7}$$

where $w_{\text{enc}} \in \mathbb{R}^{x_{\text{width}} \times x_{\text{height}}, n_{\text{features}}}$ (resp. $w_{\text{dec}} \in \mathbb{R}^{n_{\text{features}}, x_{\text{width}} \times x_{\text{height}}}$) is the weight matrix of the connections in the encoder (resp. the decoder), $\mathbf{b}_{\text{enc}} \in \mathbb{R}^{n_{\text{features}}}$ (resp. $\mathbf{b}_{\text{dec}} \in \mathbb{R}^{x_{\text{width}} \times x_{\text{height}}}$) is the bias vector of the encoder (resp. the decoder), and $f_\sigma(.)$ is some activation function[3], in this chapter, the sigmoid activation function is used $f_\sigma(x) = \frac{1}{1+e^{-x}}$. The output of the encoder corresponds to the visual features learned by the auto-encoder: $f_{\text{e}} = f_{\text{enc}}(\mathbf{X})$.

To make the auto-encoder learn useful representations, the initial approach was to impose an information bottleneck on the model, by learning representations with dimensionalities lower than the ones of the input data ($n_{\text{features}} < x_{\text{width}} \times x_{\text{height}}$). However, such low-dimensional representations cannot capture the richness of the visual information, so current approaches tend to use over-complete ($n_{\text{features}} > x_{\text{width}} \times x_{\text{height}}$) representations instead. In this case, some additional constraints must be enforced on the model to prevent it from learning trivial solutions, e.g., the identity function. These constraints generally take the form of an additional term in the objective function, for instance: weight regularization, explicit sparsity constraints (sparse auto-encoders [56], [185], k-sparse auto-encoders [63]) or regularization of the Jacobian of the encoder output $\mathbf{g}$ (contractive auto-encoders [187]). Another

[3] Only models where no activation function is applied to the decoder output are considered as the input are continuous (image) data in $[0, 1]$.

approach is to change the objective function from reconstruction to another criterion, for instance data denoising [60].

In this chapter, sparse AE are considered as a baseline to assess the performances of STDP-based feature learning. More recent models (denoising AE [60], contractive AE [187], etc.) can reach better performance, but sparse AEs are closer to current STDP-based SNNs, which also feature explicit sparsity constraints, usually through lateral inhibition. Also, it allows us to set a minimum bound that SNNs should at least reach to be competitive with regular feature learning algorithms, and identify some directions to follow to achieve this goal; it constitutes a first step before taking STDP-based SNNs further. In the following, weight regularization and sparsity constraint terms used in experiments are described:

- L2 weight regularization: $\frac{\kappa}{2}(||w_{\text{enc}}||_2^2 + ||w_{\text{dec}}||_2^2)$, where $||.||_2$ denotes the Frobenius norm and κ is the weight decay parameter;

- sparsity term [185]: $\upsilon.\text{KL}(\widehat{\rho}||\rho)$, where ρ is the desired sparsity level of the system, $\widehat{\rho}$ is the vector of average activation values of the hidden neurons over a batch, KL(.||.) the Kullback-Liebler divergence, and υ the weight applied to the sparsity term in the objective function.

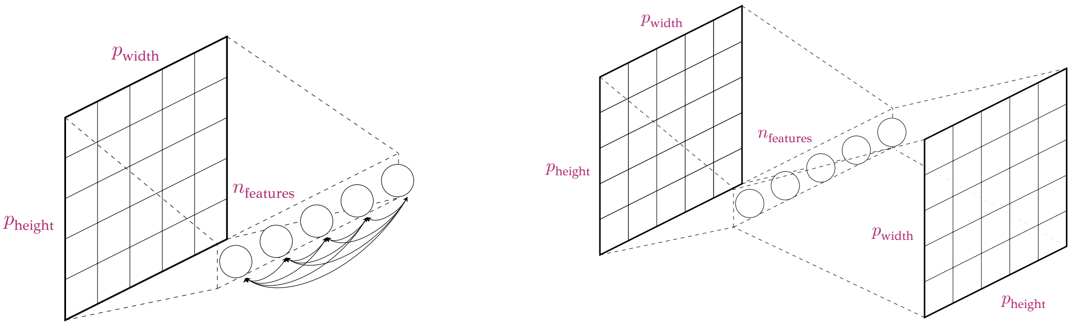
This yields the final objective function for the auto-encoder:

$$f_{\text{obj}}(\mathbf{X}, \widehat{\mathbf{X}}; \Phi) = \frac{1}{2}||\mathbf{X} - \widehat{\mathbf{X}}||_2^2 + \frac{\kappa}{2}(||w_{\text{enc}}||_2^2 + ||w_{\text{dec}}||_2^2) + \upsilon.\text{KL}(\widehat{\rho}||\rho) \tag{5.8}$$

5.4 Experiments

5.4.1 Experimental protocol

The SNN and AE architectures used in experiments are single-layer networks with n_{features} hidden units (see Figure 5.2). the experimental protocol proposed by Coates *et al.* [56] is used to compare unsupervised feature extractors. It is organized in two stages, described below: visual feature learning, and the evaluation of the learned features on image classification benchmarks.



(a) SNN

(b) Auto-encoder

Figure 5.2: (a) SNN architecture used in the experiments. Solid arrows denote inhibitory connections between hidden units. (b) AE architecture used in the experiments.

Feature learning From the training image dataset $\mathcal{X}_{\text{train}} = (\mathbf{X}_1, \mathbf{X}_2, \dots, \mathbf{X}_n)$, n_{patches} patches of size $p_{\text{width}} \times p_{\text{height}}$ are randomly samples. The patches are fed to the feature learning algorithm for training, to produce a dictionary of $|\mathbf{g}|$ features.

Image recognition The learned feature dictionary is used to produce image descriptors that are fed to a classifier following this process (Figure 5.3):

1. Image patches of size $p_{\text{width}} \times p_{\text{height}}$ are densely sampled from the images with stride l_{stride}, producing $o_{\text{width}} \times o_{\text{height}}$ patches per image (Figure 5.3a).
2. Patches are fed to the feature extractor, producing $o_{\text{width}} \times o_{\text{height}}$ feature vectors of dimension n_{features} per image, organized into feature maps where each position corresponds to one patch of the input image (Figure 5.3b).
3. Sum pooling over a grid of size $r_{\text{width}} \times r_{\text{height}}$ is applied: the feature vectors of the patches within each grid cell are summed to produce a unique vector of size n_{features} per cell. These vectors are then concatenated to produce a single feature vector of size $r_{\text{width}} \times r_{\text{height}} \times n_{\text{features}}$ for each image (Figure 5.3c).
4. The feature vectors of the images are fed to a linear SVM for training (training set) or classification (test set).

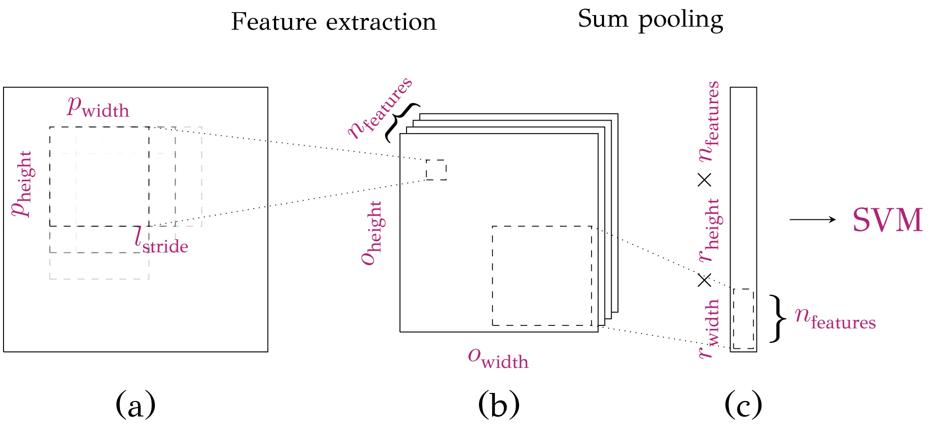


Figure 5.3: Experimental protocol. (a) Input image, where $o_{\text{width}} \times o_{\text{height}}$ patches of size $p_{\text{width}} \times p_{\text{height}}$ are extracted with a stride l_{stride}. (b) n_{features} feature maps of size $o_{\text{width}} \times o_{\text{height}}$ produced by the feature extractor from its dictionary of n_{features} features. (c) Output vector constructed by sum pooling over $r_{\text{width}} \times r_{\text{height}}$ regions of the feature maps.

5.4.2 Datasets

Experiments are performed on three datasets commonly used to evaluate unsupervised feature learning algorithms: CIFAR-10, CIFAR-100, and STL-10. Table 5.1 provides the properties of these datasets. Since previous work evaluated SNNs only on grayscale images, experiments are also performed on grayscale versions of the three datasets, referred to as CIFAR-10-bw, CIFAR-100-bw, and STL-10-bw.

Dataset	Resolution	$\lvert\mathcal{C}\rvert$	$\lvert\mathcal{X}_{\text{train}}\rvert$	$\lvert\mathcal{X}_{\text{test}}\rvert$
CIFAR-10 [75]	32 × 32	10	50,000	10,000
CIFAR-100 [75]	32 × 32	100	50,000	10,000
STL-10 [56]	96 × 96	10	5,000	8,000

Table 5.1: Properties of the datasets used in the experiments.

Contrary to MNIST, which is the preferred dataset in the SNN literature [188], these datasets provide color images of actual objects rather than just binary images

Data	$\mathbf{n}_{\text{features}}$	$\boldsymbol{\rho}$ $\boldsymbol{\upsilon}$	$\boldsymbol{\kappa}$	
Color	64	0.005	0.5	10^{-4}
	1024	0.005	0.1	10^{-5}
Grayscale	64	0.01	0.05	10^{-5}
	1024	0.005	0.1	10^{-5}

Table 5.2: AE parameters used in the experiments.

		Neuron			
$v_{\text{th}}(0)$	20 mv	v_{rest}	0 mv		
		STDP			
w_{min}	0.0	w_{max}	1.0	d_{min}	0.0
d_{max}	0.01	$\eta_{\text{w}+}$	0.001	$\eta_{\text{w}-}$	0.001
β	1.0				
		Neural Coding			
$t_{\text{exposition}}$	1.0				
		Threshold Adaptation			
t_{expected}	0.7	η_{th}	0.001		
		Pre-processing			
$\text{DoG}_{\text{center}}$	1.0	$\text{DoG}_{\text{surround}}$	2.0	DoG_{size}	7

Table 5.3: SNN parameters used in the experiments.

of digits. It makes it possible to evaluate SNNs in more realistic conditions, in terms of data richness and importance of image pre-processing. Also, unlike MNIST, but also other datasets such as NORB, they are not solved or nearly-solved problems (classification accuracy above 95%), so the results can highlight better the properties of the algorithms.

5.4.3 Implementation details

Image patches of size 5×5 pixels ($p_{\text{width}} = 5, p_{\text{height}} = 5$) and a stride $l_{\text{stride}} = 1$ are used in all the experiments. The algorithms are evaluated with two sizes of feature dictionaries, $n_{\text{features}} = 64$ and $n_{\text{features}} = 1024$. To produce final image descriptors, features are pooled over 2×2 image regions ($r_{\text{width}} = 5, r_{\text{height}} = 5$), yielding image descriptors of size $4 \times n_{\text{features}}$.

A grid search is used to find the optimal parameters for the AEs and only results for the best configuration for each experimental setting are reported. Table 5.2 provides the values of the parameters that are retained in this chapter. These parameters were consistently optimal over datasets. The AEs are trained for 1,000 epochs on 200,000 random patches from the training set considered. Adadelta optimizer [189] is used with an initial learning rate $\eta = 1.0$. AEs are implemented using TensorFlow [190]. Table 5.3 provides the parameters used to train SNNs. Since SNNs have a large number of parameters but are also time-consuming when simulated on software, only a greedy search can be used to set the parameters. The optimal value of each parameter was searched while the values of all other parameters were fixed. Thus, all the results reported in this section can slightly change due to the unfair comparison between AEs and SNNs. All SNN models are trained on 100,000 random patches from the training sets for 100 epochs. Classification was performed using LibSVM [191] with a linear kernel and default

Dataset	Color coding	$n_{\text{features}} = 64$	$n_{\text{features}} = 1024$
CIFAR-10	RGB opponent	37.66 ± 0.73	45.04 ± 0.06
	Bio-color	37.53 ± 0.33	43.54 ± 0.07
	Grayscale	45.37 ± 0.13	52.78 ± 0.41
	Grayscale + color	48.27 ± 0.47	56.93 ± 0.59
CIFAR-100	RGB opponent	17.14 ± 0.22	19.87 ± 0.03
	Bio-color	17.06 ± 0.09	19.19 ± 0.35
	Grayscale	18.43 ± 0.34	22.67 ± 0.36
	Grayscale + color	25.20 ± 0.76	30.44 ± 0.48
STL-10	RGB opponent	44.13 ± 1.30	51.20 ± 0.30
	Bio-color	44.23 ± 0.41	50.95 ± 0.08
	Grayscale	44.66 ± 0.87	51.40 ± 0.69
	Grayscale + color	49.20 ± 1.04	54.34 ± 0.30

Table 5.4: Classification accuracy (%) w.r.t. to the color coding strategy.

parameters. All reported accuracies are averaged over three runs of the feature learning algorithms.

5.4.4 Color processing with SNNs

The first experiment evaluates the strategies to encode color information in SNNs that were discussed in Section 5.2.3: images are first encoded using one of these strategies, then on-center/off-center coding is applied to each image channel. Table 5.4 shows the classification accuracies yielded by each color coding strategy on the three datasets, as well as the results obtained on grayscale images. Both color codings, biological channels (red/green, yellow/blue) or RGB opponent channels (red/green, green/blue, blue/red), provide similar recognition rates. However, using grayscale images yields better results than color images. This is a counter-intuitive result, since color images contain all the information available from grayscale images. Since the SNN processes all inputs in the same way, on-center/off-center coding should be the source of this information loss. However, this preprocessing step is currently required to extract edges from the images and feed the SNN inputs with spike trains that represent specific visual information. Training an SNN directly from RGB images could be an alternative but appears to be very challenging, because the existing mechanisms are not adapted to learn from this type of data. Notably it is difficult to find an effective threshold adaptation rule that is able to both maintain the homeostasis of the system and to add competition between neurons. One reason is that the sum of input patterns can widely vary from dark patches, where the sum is close to zero, to the bright patches, where the sum can go high. Figure 5.4 shows examples of filters learned from raw RGB images; since the network has a single layer, the filter image for one neuron can be obtained by simply interpreting the normalized weights of its input synapses as RGB values. Many filters converge towards similar or uninformative patterns. This results in large amounts of dead units and repeated features.

Finally, the last configuration is evaluated, which combines color and grayscale images by training half of the features on each input independently. Results in Table 5.4 show that this strategy provides the best performance, showing that DoG-filtered color images still contain information that grayscale DoG-filtered images do not contain. In the remaining of the chapter, this strategy is used for all the runs performed on color images.

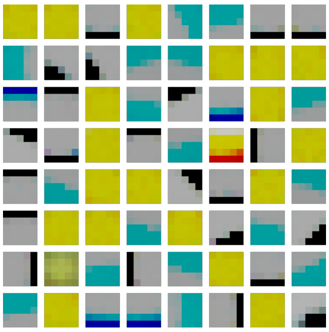


Figure 5.4: Example of SNN features learned on raw RGB pixels (trained on CIFAR-10). They are mostly dead units or simple repeated patterns.

Dataset	**SNN**		**AE**	
	$n_{\text{features}} = \mathbf{64}$	$n_{\text{features}} = \mathbf{1024}$	$n_{\text{features}} = \mathbf{64}$	$n_{\text{features}} = \mathbf{1024}$
CIFAR-10	48.27±0.47	56.93±0.59	57.56±0.08	66.98±0.33
CIFAR-10-bw	45.37±0.13	52.77±0.41	53.69±0.34	59.50±0.17
CIFAR-100	25.20±0.76	30.45±0.48	37.71±0.19	36.43±0.29
CIFAR-100-bw	18.43±0.34	22.67±0.36	23.62±0.18	26.56±0.05
STL-10	49.20±1.04	54.34±0.30	52.28±0.47	55.74±0.25
STL-10-bw	44.66±0.87	51.40±0.69	50.63±0.23	52.88±0.29

Table 5.5: Average classification accuracy (%) and its standard deviation w.r.t. to the datasets and feature learning algorithms.

5.4.5 SNNs versus AEs

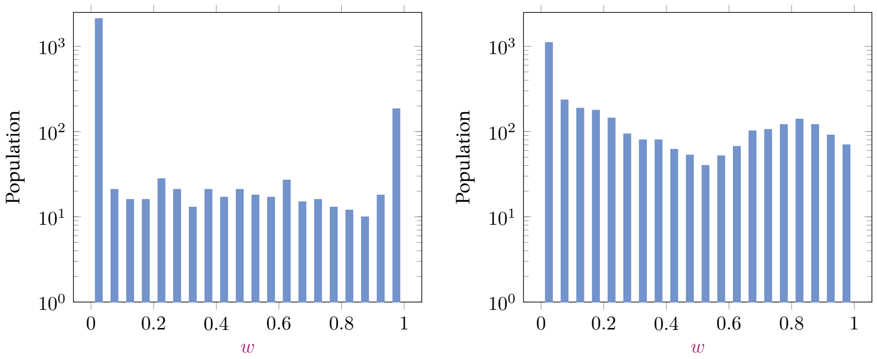
The classification accuracies for each feature learning algorithm and dataset are reported in Table 5.5. AEs perform consistently better than SNNs[4]. So, how to bridge the gap between STDP learning and standard neural network approaches? Several elements may explain the performance of STDP. The results reported in Table 5.5 show two trends. First, working with colors always yields better results than working with grayscale images; a straightforward explanation is that color is significant to recognize objects in the datasets considered, either because natural objects (e.g. animals) represented in the datasets have a limited, meaningful set of colors, either because the contexts of the objects (e.g. the sky behind airplanes) have meaningful colors. The second trend is that the performance gap between SNNs and AEs is larger on color images than on grayscale images, showing that SNNs cannot handle color well, at least not with the straightforward color codings that were used in the experiments. This result highlights the importance of color in object recognition, and therefore the need for more efficient neural codings of color in SNNs.

Looking at the filters learned by SNNs and AEs provides additional information about the properties of features learned by STDP and potential reasons for the performance gap. Figures 5.6 and 5.7 show samples of filters learned by SNNs and AEs, respectively. The filters are different in nature. Filters learned by STDP are mostly edges, and some blobs, that are well-defined, with one or two dominant colors. By contrast, AEs learn more complex features; edges and blobs can still be observed, but they include a larger range of color or gray levels and are not as

[4] As stated in Section 5.3, a simple sparse auto-encoder is used rather than a state-of-the-art model. It means that the actual gap between SNNs and state-of-the-art feature learning methods would be larger than what these experiments show.

Dataset	$\beta = 1$	$\beta = 2$	$\beta = 3$	$\beta = 4$
CIFAR-10	48.27±0.47	46.56±0.68	43.18±1.60	41.03±0.21
CIFAR-10-bw	45.37±0.13	44.55±0.57	41.74±1.50	38.90±1.57

Table 5.6: SNN recognition rate according to STDP β parameter ($n_{\text{features}} = 64$).



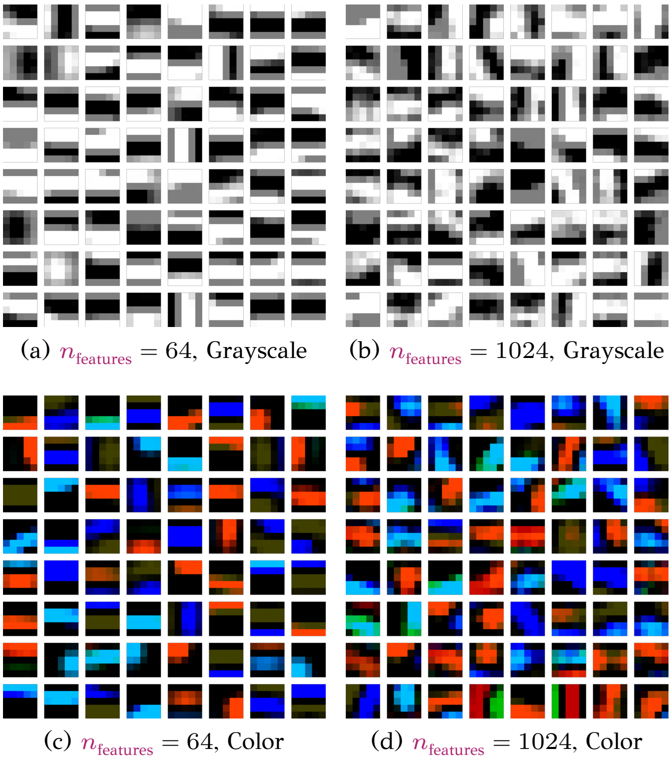
(a) β = 1.0

(b) β = 4.0

Figure 5.5: Distribution of weights (log. scale) in an SNN ($n_{\text{features}} = 64$) after training w.r.t. β. Most weights have values close to 0 or 1 when β decreases.

elementary as the ones learned by SNNs. Simple, well-defined features like the ones learned by STDP are conceptually pleasing because they represent elementary object shapes that can easily be understood. They suggest better generalization abilities from the feature extractor, and correspond to biological observations [151]. However, they are not as effective in practice. AEs can also produce features closer to the ones obtained with SNNs (although with larger ranges of tones and intensities), but such features are obtained only by increasing the weight of L2 regularization, usually at some cost in accuracy. The specific looks of SNN features can be explained in two ways. First, the use of on-center/off-center coding as a preprocessing step biases the learning algorithms towards edge-like filters, as it highlights the edges in the images.

Moreover, the fact that the learned features contain exclusively black or saturated colors is due to the fact that STDP rules tend towards a saturating regime for weights: once a given unit has learned a pattern, repeated expositions to this pattern will reinforce the sensitivity to this pattern until the weights reach either 1 or 0. This is illustrated in Figure 5.5a, which shows the distribution of weights in an SNN after training: most weight values are close to 0 or 1. Since AEs perform better and have more staggered weights, one may believe that saturated weights are detrimental to the performance of SNNs. To check this, experiments are performed with different values for the parameter β: increasing their values allow the weights to ”escape” more easily from their limit values w_{min} and w_{max}. Figure 5.5b shows that the weights are indeed more staggered, but the classification accuracy decreases as β get larger (see Table 5.6). The fact that STDP leads to saturated weights may not be the only reason for the performance gap with AEs. Finally, the filters shown in Figure 5.6 also show a good property of SNNs: they do not raise any dead units, i.e. features that get stuck in a state with average weights that do not correspond to any significant pattern. By contrast, AEs tend to learn a fair amount of such features, especially when the number of features increases (see Figure 5.7). This behavior of SNNs can be due to two factors: lateral inhibition, which prevents neurons from learning similar patterns (here, becoming dead units), and the saturated regime of STDP.



(a) $n_{\text{features}} = 64$, Grayscale (b) $n_{\text{features}} = 1024$, Grayscale

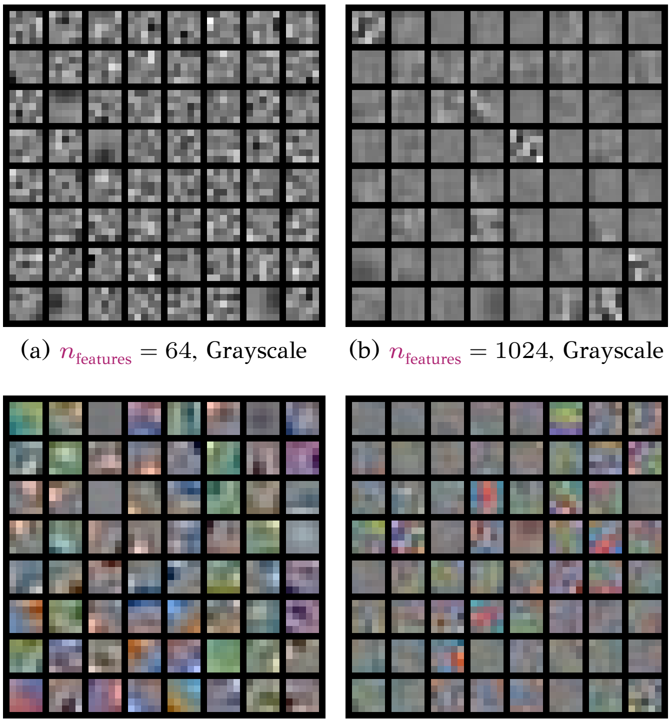
(c) $n_{\text{features}} = 64$, Color (d) $n_{\text{features}} = 1024$, Color

Figure 5.6: Grayscale and color filters learned by SNNs on CIFAR-10-bw and CIFAR-10. For $n_{\text{features}} = 1024$, random samples are shown.

5.5 Result Analysis and Properties of the Networks

5.5.1 On-center/off-center coding

In this section, the impact of on-center/off-center coding on classification accuracy is investigated. As mentioned in Section 5.4.5, this image coding is responsible for the type of visual features learned by STDP, but does it impact the final accuracy of the system? The accuracy of two systems are compared, each with and without preprocessing images: an AE, under the same protocol as before, and an SVM performing classification directly from image pixels. The AE parameters for the on-center/off-center coding runs are: $\rho = 0.005$, $\upsilon = 1.0$, and $\kappa = 10^{-4}$. Results on CIFAR-10 and CIFAR-10-bw are reported in Table 5.7. Using on-center/off-center coding decreases the accuracy of the classification in both configurations, which confirms that this coding is one of the causes of the limited performance of SNNs in image classification. This is due to the fact that extracting edges with DoG has the effect of selecting only a subrange of spatial frequencies. In addition, the accuracies obtained on filtered color images are only on par with (in the case of AEs) or worse than (with SVM) the results obtained using grayscale images; it highlights the fact that on-center/off-center coding cannot handle color effectively. One reason is that edge information is effectively represented by grayscale pixels, and the additional information brought by color is essentially located in uniform image regions. Interestingly, the unsupervised SNN models of the literature that are competitive with traditional approaches are only evaluated on the MNIST dataset [188], which does not require on-center/off–center coding as the images are only made of edges (white handwritten digits on black backgrounds). Therefore, to



(a) $n_{\text{features}} = 64$, Grayscale (b) $n_{\text{features}} = 1024$, Grayscale

(c) $n_{\text{features}} = 64$, Color (d) $n_{\text{features}} = 1024$, Color

Figure 5.7: Grayscale and color filters learned by AEs on CIFAR-10-bw and CIFAR-10. For $n_{\text{features}} = 1024$, random samples are shown.

Dataset	**Raw pixels**	**AE features** $n_{\text{features}} = 64$
CIFAR-10	37.79	57.56±0.08
CIFAR-10-dog	21.07	52.65±0.30
CIFAR-10-bw	28.38	53.69±0.34
CIFAR-10-dog-bw	25.29	52.76±0.08

Table 5.7: Classification accuracy (%) obtained with raw pixels and AE features w.r.t. pre-processing methods. Only one run is performed on raw pixels as SVM training is deterministic.

be effective, SNNs require the design of a suited image coding that preserves as much visual information as possible. Using alternative methods to extract edges (such as the image gradient or the image Laplacian) could capture slightly different types of edge information, which could be processed within a single SNN for improved performance, in a feature fusion approach. However, this would only process edge information, which is insufficient to reach optimal classification performances. Ideally, SNNs should be able to handle raw RGB pixels in order to get all available information; however, this is not straightforward, as showed in Section 5.4.4.

5.5.2 Sparsity

An investigation of the sparsity properties of SNNs and AEs is done here. To do so, the following sparseness measure [192] is used:

$$\mathrm{sp}(\mathbf{g}) = \frac{\sqrt{n_{\text{features}}} - \frac{\sum_i^{n_{\text{features}}} |g_i|}{\sqrt{\sum_i^{n_{\text{features}}} g_i^2}}}{\sqrt{n_{\text{features}}} - 1} \tag{5.9}$$

where $\mathbf{g}$ is the vector of activations of hidden units (i.e. the visual feature vector) and n_{features} is the number of hidden units. $\text{sp}(\mathbf{g}) \in [0, 1]$; larger values indicate sparser activations.

Table 5.8 shows the mean sparseness of features computed on the test set of CIFAR-10. The sparseness is much higher in SNNs than in AEs. Indeed, the specialization of features in SNNs relies mostly on lateral inhibition, which prevents units from integrating spikes, leading to very sparse activations of the features. Sparsity is often cited as a necessary condition for good representations [55], and has been shown to be correlated to classification accuracy on image datasets [185]. However, some results in [185] show that maximizing sparsity does not always lead to improvements in classification accuracy in AEs. Similarly, enforcing too much sparsity on the AEs (e.g., by lowering ρ) is detrimental to the classification accuracy. To push it further, five runs on CIFAR-10 are performed with $n_{\text{features}} = 64$ and different values for parameters κ, υ, and ρ. The AE parameters were set so that the sparseness would be close to the sparseness that was measured in SNNs (i.e., in the range $[0.8; 0.9]$). In these runs, the classification accuracy varies from 35.53% to 41.03%, much lower than the 57.56% baseline. To check whether high levels of sparseness are an issue for SNNs too, an experiment where lateral inhibition is deactivated during the feature extraction phase is run.

As expected, deactivating inhibition decreased the sparseness of the model (from 0.869 to 0.638 on CIFAR-10). However, the classification rate decreased too (from 48.27% to 47.35%). It shows that, although sparsity is a desirable feature for good representations, an excessive level of sparseness can be detrimental, and that the right amount of sparsity should be enforced during training. This calls for the use of other, less restrictive, inhibition strategies than WTA.

Model	$\mathbf{n}_{\text{features}} = \mathbf{64}$	$\mathbf{n}_{\text{features}} = \mathbf{1024}$
SNN	$0.869 \pm 1.96\text{e}{-5}$	$0.967 \pm 3.04\text{e}{-5}$
AE	0.352 ± 0.116	0.112 ± 0.077

Table 5.8: Mean and standard deviation of feature sparseness (test set of CIFAR-10).

5.5.3 Coherence

One measure of the quality of the learned feature is their incoherence, i.e. the fact that one feature cannot be obtained by a sparse linear combination of other features in the vocabulary. If the incoherence is low, features are redundant, which is harmful for classification as redundant features will overweight other features. Inspired by the measure introduced in [63], the coherence μ_{ij} of two features g_i and g_j is measured as their cosine similarity:

$$\mu_{ij} = \frac{| < g_i, g_j > |}{||g_i||_2 . ||g_j||_2} \tag{5.10}$$

where g_i is the i^{th} feature, $< ., . >$ is the dot product operator, and $||.||_2$ is the L2 vector norm. $\mu \in [0, 1]$; 0 corresponds to orthogonal (incoherent) features and 1 to similar (coherent) features. The weights span different ranges of values depending of the feature extractor considered; feature normalization makes coherence measures comparable from one extractor to another.

Table 5.9 displays the mean and the standard deviation of coherence measure μ under all experimental settings. Overall, STDP-based SNNs produce more coherent features, which is one of the factors that can explain their lower performance, since there is a smaller variety of filters. Moreover, the maximum pairwise

Dataset	SNN		AE	
	$n_{\text{features}} = 64$	$n_{\text{features}} = 1024$	$n_{\text{features}} = 64$	$n_{\text{features}} = 1024$
CIFAR-10	0.252±0.252	0.285±0.249	0.154±0.144	0.145±0.109
CIFAR-10-bw	0.313±0.271	0.340±0.234	0.119±0.138	0.225±0.161
CIFAR-100	0.256±0.230	0.289±0.238	0.154±0.149	0.143±0.199
CIFAR-100-bw	0.320±0.238	0.343±0.223	0.121±0.137	0.234±0.166
STL-10	0.263±0.293	0.293±0.246	0.177±0.164	0.151±0.114
STL-10-bw	0.263±0.293	0.293±0.246	0.119±0.132	0.236±0.169

Table 5.9: Mean and standard deviation of feature coherence μ under all experimental settings.

coherence between two SNN features is higher ($\max(\mu_{ij}) = 0.999$ in most experimental configurations) than the maximum coherence between AE-produced features ($\max(\mu_{ij}) \in [0.898, 0.998]$), i.e. SNNs can learn almost identical features; in AEs, such features mostly correspond to dead units, whereas in SNNs they are significant features that are repeated. This result shows the limits of WTA inhibition, which should prevent features from reacting to the same patterns but fails to do so in practice. This calls for more work on understanding inhibition mechanisms and designing inhibition models that better prevent the co-adaptation of features.

5.5.4 Objective Function

One issue with STDP learning is that the objective function optimized by the system is not explicitly expressed, unlike AEs, which minimize reconstruction error. Identifying the criteria that are optimized by STDP rules would help to better understand the related learning process and design learning rules for specific tasks. This section check whether STDP rules embed reconstruction as a training criterion, by investigating if features learned through STDP are suited for image reconstruction, as those learned by AEs do.

To do so, the test images is reconstructed from the visual features. First, individual patches are reconstructed: in AEs, the reconstructed patches are directly provided by the decoder; in SNNs, patches are reconstructed as a linear combination of the filters weighted by their activations for the current sample, like in an AE with tied weights. Images are reconstructed from patches by averaging the values of overlapping patches at each location.

Table 5.10 shows the reconstruction error of each feature extractor on the test set of CIFAR-10, computed as the sum of squared errors between input images and reconstructed images, averaged over the samples. The reconstruction error is much higher for SNNs than AEs, which suggests that STDP does not learn features that allow reconstruction. However, qualitatively, the results look different (see samples in Figure 5.8): the edges of the objects are reconstructed, although with less details than in the original images, but the global illumination is degraded. The degradation of pixel intensities explains for a large part the increased reconstruction error. This is best illustrated by the best and worst reconstructions (in the sense of the mean squared error (MSE)) that are obtained using SNNs (see Figure 5.9): edges are reconstructed correctly in both, but not pixel intensities. The reason for this is that SNNs process DoG-filtered images, in which color intensities are discarded and only edge information is retained. One could expect the reconstruction error of SNNs to be much lower if they were able to process raw images directly. Also, the lack of details around the edges could be blamed on the learned features being too elementary and sparse, which prevents the reconstruction of complex patterns.

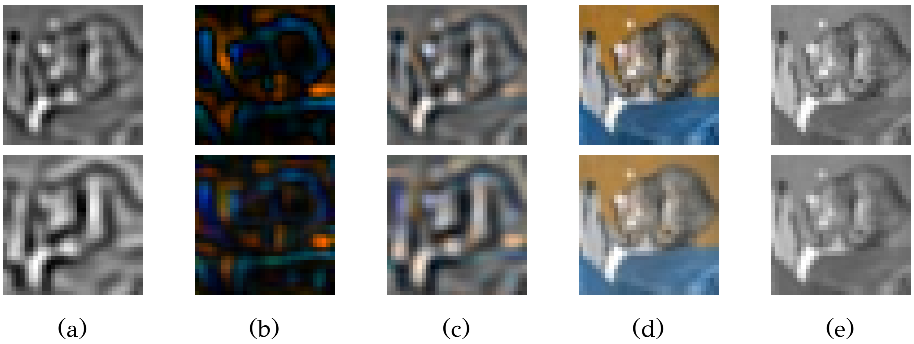


Figure 5.8: Image reconstruction samples from the test sets of CIFAR-10 and CIFAR-10-bw (top: pre-processed input images, bottom: reconstructed images). (a) SNN features, DoG-filtered grayscale image (b) SNN features, DoG-filtered color image (c) SNN features, grayscale and color DoG-filtered image (d) AE filters, color image (e) AE filters, grayscale image.

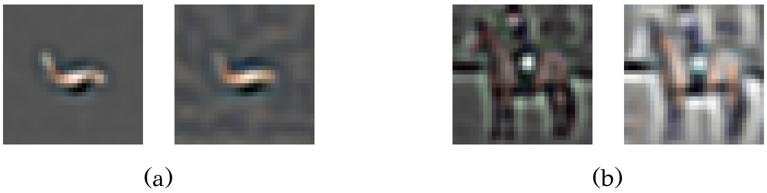


Figure 5.9: (a) Best (error: 1.60) and (b) worst (error: 15.38) reconstructions from SNN features from the test set of CIFAR-10 (left: input images, right: reconstructions).

These results show that, although this is not explicit in the learning rules, STDP learns to reconstruct images, among other potential criteria. However, it is known that minimizing reconstruction error is not sufficient to provide meaningful representations [60]. This is why recent AE models include additional criteria such has sparsity penalties [185] or Jacobian regularization [187]. How such criteria could be implemented within STDP rules, as well as which other criteria are already embedded in the STDP rules, are still open questions. Some work already show that STDP can behave similarly to the independent component analysis (ICA) [193], the principal component analysis (PCA) [194] and the non-negative matrix factorization (NNMF) [110].

5.5.5 Using Whitening Transformations with Spiking Neural Networks

As shown previously in Section 5.4.4, it is necessary to apply some pre-processing in order to learn useful features with natural colored images. Using on/off filtering

Dataset	**SNN**		**AE**	
	$n_{\text{features}} = 64$	$n_{\text{features}} = 1024$	$n_{\text{features}} = 64$	$n_{\text{features}} = 1024$
CIFAR-10	4.9429	4.4179	0.0802	0.0742
CIFAR-10-bw	4.9797	4.4628	0.00407	0.00472

Table 5.10: Average reconstruction errors on the test set of CIFAR-10.

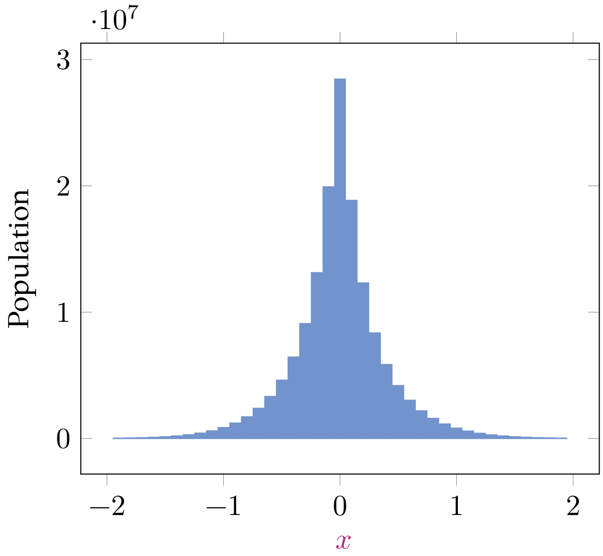


Figure 5.10: Histogram of the values after applying ZCA transformation on CIFAR-10.

helps to improve the performance, but remains an unsatisfactory solution to achieve good results. This is due to the information loss of this pre-processing. One solution can be to use multi-scale on/off filtering, in order to increase the number of spatial frequencies retained [195]. However, this alternative leads to a rapid growth of the network because connections, and maybe neurons, need to be added for each scale. A second solution is to use data whitening. This transformation leads to centered, normalized, and decorrelated data. This pre-processing method has already shown that it can improve the performance of traditional methods [196]. Zero component analysis (ZCA) consists of finding a matrix $\mathbf{W}_{\text{whiten}}$ that can be applied to data $\mathbf{X}$ to get the whitened data $\mathbf{X}_{\text{whiten}}$:

$$\mathbf{X}_{\text{whiten}} = \mathbf{W}_{\text{whiten}}\mathbf{X} \tag{5.11}$$

$\mathbf{W}_{\text{whiten}}$ can be computed from the eigenvectors and the eigenvalues of the covariance matrix computed from $\mathcal{X}_{\text{train}}$:

$$\mathbf{\Sigma} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{-1} \tag{5.12}$$

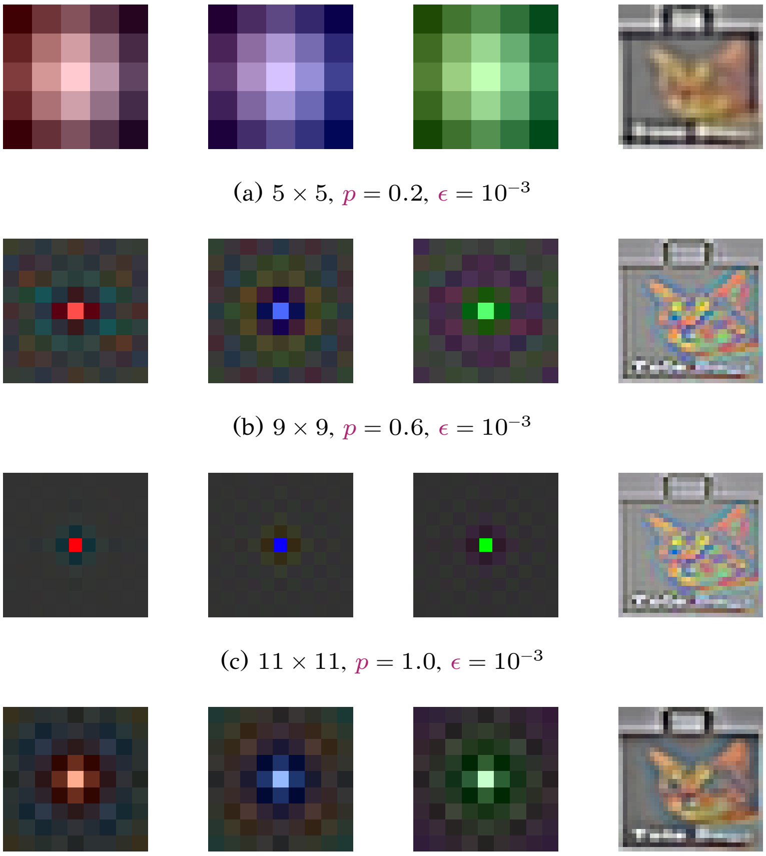
with $\mathbf{\Sigma}$ the covariance matrix, $\mathbf{U}$ the eigenvectors matrix, and $\mathbf{\Lambda}$ the diagonal matrix of eigenvalues ($\mathbf{\Lambda} = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$). Parameter $p \in [0, 1]$ is the ratio of the largest eigenvalues that are retained (i.e. the last remaining eigenvalues are set to 0, as in a PCA compression). Then, $\mathbf{W}_{\text{whiten}}$ is computed following Equation 5.13 so that the covariance of the transformed data $\mathbf{\Sigma}$ is $\mathbf{I}$, the identity matrix:

$$\mathbf{W}_{\text{whiten}} = \mathbf{U}\sqrt{(\mathbf{\Lambda} + \epsilon)^{-1}}\mathbf{U}^T \tag{5.13}$$

with ϵ the whitening coefficient, which adds numerical stability and acts as a low pass filter.

However, converting directly the whitened data $\mathbf{X}_{\text{whiten}}$ into spikes with latency coding does not allow to learn effective features, which leads to very low classification rates (10%). But by converting extreme values (see Figure 5.10) as spikes with the earliest timestamps (i.e. which represent the highest values in latency coding), performances are greatly improved (54.95%). To do so, the whitened data $\mathbf{X}_{\text{whiten}}$ are scaled in $[-1, 1]$, and then separated into two channels (one for positive values and the other for negative values, like with on/off filtering):

$$\begin{aligned} x_{\text{on}} &= \max(0, x) \\ x_{\text{off}} &= \max(0, -x) \end{aligned} \tag{5.14}$$



(a) 5×5, $p = 0.2$, $\epsilon = 10^{-3}$

(b) 9×9, $p = 0.6$, $\epsilon = 10^{-3}$

(c) 11×11, $p = 1.0$, $\epsilon = 10^{-3}$

(d) 9×9, $p = 1.0$, $\epsilon = 10^{-1}$

Figure 5.11: Examples of filter generated from ZCA transformation.

However, applying the whitening transformation is computationally expensive and is not easily implementable on neuromorphic architectures because it requires a dot product over the dimensionality of the data[5]. To bypass this issue, we propose to replace the DoG kernel used in on/off filtering by a kernel computed with the whitening transformation. This convolution kernel can be implemented in a way similar to on/off filtering and can work on different sizes of inputs. To achieve this, ZCA transformation is computed from patches of size $p_{\text{width}} \times p_{\text{height}}$ extracted from the target dataset, where $p_{\text{width}} = \text{DoG}_{\text{size}}$. It is now necessary to transform the whitening matrix $\mathbf{W}_{\text{whiten}}$ of dimension $[p_{\text{width}} \times p_{\text{height}}, p_{\text{width}} \times p_{\text{height}}]$ into three kernels (one for each RGB channel) with the same dimensions as the DoG filters ($[\text{DoG}_{\text{size}}, \text{DoG}_{\text{size}}]$). An input matrix $\mathbf{X}$ is created for each RGB channel with a single value set to 1:

$$\mathbf{X}_{i,j,k,c} = \begin{cases} 1 & \text{if } k = c \text{ and } i = \frac{\text{DoG}_{\text{size}}}{2} \text{ and } j = \frac{\text{DoG}_{\text{size}}}{2} \\ 0 & \text{otherwise} \end{cases} \tag{5.15}$$

with i, j, and k the coordinates in matrix $\mathbf{X}$ and c the current RGB channel (Red=1, Green=2, Blue=3). The dot product of the whitening matrix and these input matrices ($\mathbf{W}_{\text{whiten}}\mathbf{X}$) approximates the whitening transformation for each RGB channel. Examples of kernels generated by this method and the resulting filtered images are shown in Figure 5.11.

Some preliminary results are already available. However, these studies do not provide an exhaustive search over the different parameters, but only evaluate

[5] The whitening transformation is even more expensive to compute than to apply to the data because its requires to compute a matrix decomposition. However, this issue is not addressed in this work.

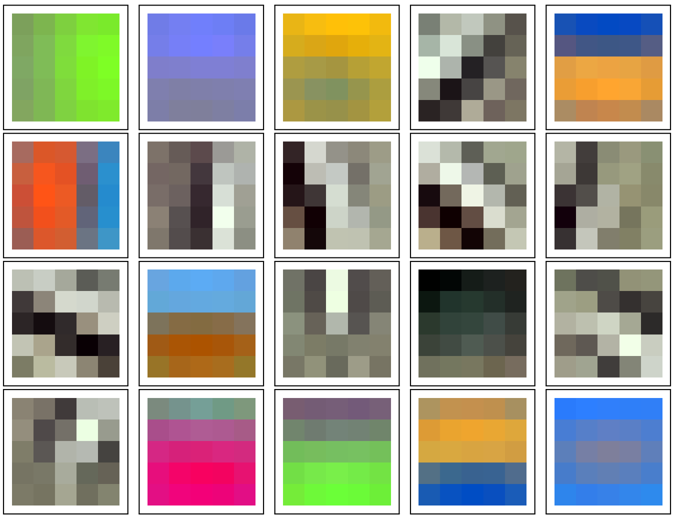


Figure 5.12: Example of filters learn with multiplicative STDP ($\beta = 3.0$) on whitened CIFAR-10.

the impact of a subset of parameters when others are arbitrarily fixed. The first experiments study the impact of the whitening transformation on the single convolution layer of a SNN with $n_{\text{features}} = 128$. Figure 5.12 shows examples of filters learned with a multiplicative STDP. These filters are closer to the ones learned with AEs (see Figure 5.7). Table 5.13 and Table 5.14 shows the impact of p according to the selected patch size with respectively multiplicative STDP and biological STDP. Removing the lasts eigenvectors can slightly improve the results. An explanation to these results may be that retaining only the most important eigenvalues forces STDP to learn the most important feature of the images. Moreover, the larger dimension of patches gives better results. Figure 5.11 shows that small patch size results in a blurred whitened image, which can explain this difference of performances.

Table 5.15 focuses on the impact of the ϵ hyper-parameter. Using 10^{-2} gives the best result (65.03%). This value removes many of the high spatial frequencies, which can help again STDP to focus on important patterns. Very low value for ϵ (e.g. 10^{-5}) degrades the results, maybe due to the greater numerical instability.

Finally, Table 5.16 shows that late t_{expected} (0.8-0.9) leads to the best performance, with 64.25% with $t_{\text{expected}} = 0.825$. We also tried to measure the performance of a three-layered SNN (We use the protocol described in Chapter 6 to train the multi-layered network). On CIFAR-10, the performances are slightly improved (from 63.28% for the first convolution to 66.58% on the second convolution). However, the results are different on STL-10 for an unknown reason: the performance decreases at the output of the second convolution layer. These observations show the benefit of using whitened data as an input to SNNs but calls for more studies in the case of multi-layered networks.

5.6 Conclusion

In this chapter, SNNs equipped with STDP and AEs are compared for unsupervised visual feature learning. Experiments on three image classification datasets showed that STDP cannot currently compete with classical neural networks trained with gradient descent, but also highlighted a number of properties of SNNs and provided specific directions towards effective feature learning with SNNs. Specifically, this chapter showed that:

- STDP-based SNNs are unable to deal naturally with RGB images; some

Patch size	**p**				
	0.2	**0.4**	**0.6**	**0.8**	**0.8**
5×5	55.25	56.70	58.95	59.61	59.50
7×7	56.42	59.10	59.77	60.35	59.45
9×9	57.62	58.97	59.72	60.22	60.04
11×11	56.97	59.05	59.84	**60.45**	59.90

Figure 5.13: Recognition rates on CIFAR-10 with a multiplicative STDP ($\beta = 3.0$), using ZCA whitening according to the ratio of eigenvectors p used and the patch size ($\epsilon = 0.1$, $t_{\text{expected}} = 0.85$).

Patch size	**p**				
	0.2	**0.4**	**0.6**	**0.8**	**0.8**
5×5	57.09	61.94	63.65	63.70	63.20
7×7	59.97	63.96	93.91	63.17	63.34
9×9	62.34	**64.57**	63.52	63.9	63.94
11×11	62.11	64.39	64.53	64.05	54.53

Figure 5.14: Recognition rates on CIFAR-10 with a biological STDP ($\tau_{\text{STDP}} = 0.1$), using ZCA whitening according to the ratio of eigenvectors p used and the patch size ($\epsilon = 0.1$, $t_{\text{expected}} = 0.85$).

Patch size	ϵ				
	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}
5×5	61.93	64.14	62.42	60.28	54.04
7×7	62.76	64.64	63.84	60.24	55.01
9×9	63.33	**65.03**	63.82	60.00	54.85
11×11	64.00	64.33	62.84	60.78	54.45

Figure 5.15: Recognition rates on CIFAR-10 with a biological STDP ($\tau_{\text{STDP}} = 0.1$), using ZCA whitening according to the whitening coefficient ϵ and the patch size ($p = 1.0$, $t_{\text{expected}} = 0.85$).

$\mathbf{t}_{\text{expected}}$	**CIFAR-10**		**SLT-10**	
	conv1	**conv2**	**conv1**	**conv2**
0.700	61.85	60.91	51.1857	44.2500
0.725	61.52	64.63	51.8125	44.7125
0.750	63.12	65.56	52.0125	47.1875
0.775	63.40	65.98	54.8875	50.6000
0.800	63.12	65.61	55.2875	50.3125
0.825	64.25	65.43	54.9875	50.8500
0.850	64.10	66.35	55.5125	49.4000
0.875	64.30	65.88	55.1750	51.1875
0.900	63.28	**66.58**	54.8750	50.2500
0.925	62.40	65.88	52.2750	48.5000
0.950	57.90	58.47	47.8000	44.3000
0.975	53.21	49.55	44.4000	39.6000

Figure 5.16: Recognition rate on CIFAR-10 with a biological STDP ($\tau_{\text{STDP}} = 0.1$), using ZCA whitening according to t_{expected} ($p = 1.0$, size = 9×9, $\epsilon = 10^{-2}$).

pre-processing of the images is required to learn significant visual features.

- the common on-center/off-center image coding used in SNNs results in an information loss, thus harming the classification accuracy; this information loss is even more pronounced on color images;
- WTA inhibition results in overly sparse features and does not prevent the co-adaptation of features in practice;
- STDP-based learning rules produce features that enable to reconstruct images from the learned features, as AEs do, even though the features are not explicitly optimized for this task. However, the quality of the reconstruction is harmed by the limitations of the model.

Whitening is a solution to avoid the loss of information encountered with on/off filtering. Preliminary results show the potential of this method. However, more work remains necessary to offer mechanisms that approximate this transformation while remaining implementable on neuromorphic architectures in an energy-efficient way. Another direction is to design inhibition rules that promote distinctive patterns and enforce the right level of activity sparsity; such rules should be "soft", i.e. allow more than one neuron to spike at once. Methods that control the level of sparsity in AEs, as in [63], are good candidates but should be adapted to preserve the locality of computations, which is a major asset of STDP-based SNNs. The next chapter investigates different mechanisms of SNNs, such as the inhibition, the threshold adaptation, and the STDP rule, in order to make effective multi-layered SNNs. To this end, the threshold adaption rule introduced in this part will be slightly modified.

Chapter 6

Training Multi-layer SNNs with STDP and Threshold Adaptation

Using deep hierarchical representations improves the expressiveness of models [46]. However, setting up a multi-layered SNN trained with STDP remains a challenge, and only little work succeeds in providing effective models [150], [151]. One reason is that SNN performances are highly sensitive to the model parameters. Since SNNs have a large number of parameters and the simulation of these models is time-consuming, performing an exhaustive search is not yet possible. Thus, the parameters optimization step is laborious and non optimal explaining the difficulties of getting SNNs that can compete with traditional methods. Reducing the impact of parameter values, by using auto-adaptive parameters, or at least, reducing the number of parameters, seems to be a key point in order to be able to make SNNs viable. This chapter extends mechanisms developed in the previous chapter in order to allow learning features in a multi-layer fashion. We modify the threshold adaptation mechanism in order to improve the performance of the network (see Section 6.2). Additionally, we propose a protocol to train multi-layered networks. We experiment with multi-layered SNNs on the Faces/Motorbikes [151] and MNIST [65] datasets and carry out multiple studies to evaluate the impact of the threshold adaptation system, but also of the inhibition policy and of the STDP rule. Finally, we test the combination of multiple networks trained with different parameters to improve the classification rate thanks to the different patterns learned by the network.

6.1 Network Architecture

The networks used in this chapter are composed of stacked FF layers. As in the previous chapter, IF neurons are used in order to reduce the number of parameters. Moreover, on/off filtering (i.e. only grayscale images are used), but also temporal coding, are used in order to convert images into spikes train. For a layer $l(n)$, there are $l_{\text{depth}}(n)$ feature maps, each of them containing $l_{\text{width}}(n) \times l_{\text{height}}(n)$ neurons. Three types of layers are used in this chapter: convolution, pooling, and fully-connected layers. In the pooling layers, all the parameters are constant: neuron thresholds and synaptic weights are fixed to 1. When a spike is triggered in its receptive field, a pooling neuron directly fires a spike. This mimics a max-pooling operation. A column $q_{x,y}(n)$ designates the $l_{\text{depth}}(n)$ neurons present at position (x, y) in the $l_{\text{depth}}(n)$ features maps of $l(n)$.

6.2 Training Multi-layered Spiking Neural Networks

The mechanisms used in the multi-layered SNNs are similar to these introduced in the previous chapter. However, in order to simplify the model, no delay is used in the network. This restriction allows to increase the parallelism in the model and reduce the number of parameters. However, delays may play a major role in the learning of temporal patterns. Section 6.2.1 extends the threshold adaptation rule used in Section 5.2.1. Section 6.2.2 offers a spike-to-value conversion function to interpret the output of the network and Section 6.2.3 describes the protocol used to train multi-layered SNNs.

6.2.1 Threshold Adaptation Rule

The threshold adaptation rule described in Section 5.2.1 is reused. However, a new parameter $\mathrm{th}_{\mathrm{min}}$ is added, which limits the minimum value that the threshold v_{th} can take. This constraint forces the neurons to integrate a minimum number of spikes, and so, to reinforce a sufficient number of connections. This parameter is usefull in some case (see Section 6.3.3) to ensure that neurons learn effective patterns. Thus, the threshold update equation becomes:

$$v_{\mathrm{th}}(t) = \max(\mathrm{th}_{\mathrm{min}}, v_{\mathrm{th}}(t-1) + \Delta^1_{\mathrm{th}} + \Delta^2_{\mathrm{th}}) \tag{6.1}$$

with Δ^1_{th} and Δ^2_{th} the threshold update computed respectively by Equation 5.2 and Equation 5.2 and 5.3.

WTA inhibition drastically reduces the spiking activity, which can lead to poor classification performances (see Section 5.5.2). For this reason, the inhibition mechanism is removed during the inference stage (i.e. for training next layers or for generating the feature vector $\mathbf{g}$). An intermediate inhibition policy, named soft inhibition, is also investigated in this chapter. This policy uses inhibition spikes, which reduces the membrane voltage v of the other neurons by a v_{inh} constant, but does not prevent them from firing.

6.2.2 Network Output

A new method is used to interpret the output of the network to take into account the parameter t_{expected}. Since latency coding is used, the earliest output spikes will encode the highest values. Output values y are computed according to the expected t_{expected} set in the output layer, following this equation:

$$y = \min\left(1, \max\left(0, 1 - \frac{t - t_{\mathrm{expected}}}{t_{\mathrm{exposition}} - t_{\mathrm{expected}}}\right)\right) \tag{6.2}$$

with t the spike timestamp (set to $+\infty$ if no spike occurs).

6.2.3 Training

Traditionally, convolution requires to perform non-local operations and to use non-local memory since they use shared weights: columns need to communicate with each other to share the same filters. A specific training protocol is used in order to reduce the cost of the global communication needed by the convolutions. One layer is trained at a time, from the layer closest to the input to the one at the output of the network. During the training of a convolution layer, only one column is activated to discard the usage of inter-column communications. Once the layer is trained, its parameters (weights and thresholds) are fixed and are copied onto the other columns of the layer. This operation is necessary since pooling

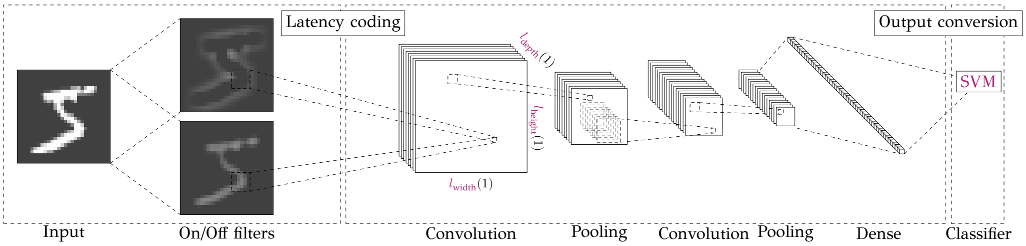


Figure 6.1: Network topology

layers require the same filters in adjacent columns. In order to keep the position invariance brought by shared weights, random patches of size $h_{\text{width}}(n) \times h_{\text{height}}(n)$ are extracted from inputs of the layer. Unlike in [151], neurons do not react only to the most salient part of each image.

6.3 Results

6.3.1 Experimental protocol

The protocol described in Section 5.4 is adapted to multi-layered SNNs. For each trained layer, the training set is processed n_{epoch} times. A simulated annealing procedure is applied after every epoch: the learning rates (i.e. η_{w} and η_{th}) are decreased by a factor α. This helps to converge to a stable state during the training. Once the training is finished, the training set and the test set are processed by the network, which converts all the samples into their output representation. If the output layer has multiple columns (i.e. $l_{\text{width}}(n) > 1$ or $l_{\text{height}}(n) > 1$), sum-pooling is applied over the positions of the feature maps to produce a feature vector $\mathbf{g} = (g_1, \cdots, g_x)$:

$$y_k = \sum_{i=0}^{o_{\text{width}}} \sum_{j=0}^{o_{\text{height}}} y_{ijk} \tag{6.3}$$

with y_{ijk} the value of output of the network at position (i, j) in feature map $k \in [0, o_{\text{depth}}]$. If the output layer has only one column, it directly outputs vector $\mathbf{g}$. An SVM with a linear kernel is trained over the output training set. SVM parameters are not optimized ($\text{svm}_c = 1$). Figure 6.1 shows the complete network topology. Besides classification rates, the sparsity of the network is also investigated. The sparsity is computed over the output vectors $\mathbf{g}$ of the test set with the following formula, used in Section 5.5.2.

All the results reported in this chapter are averaged over 10 runs. The default parameters are reported in Table 6.1.

6.3.2 MNIST

Threshold Target Time

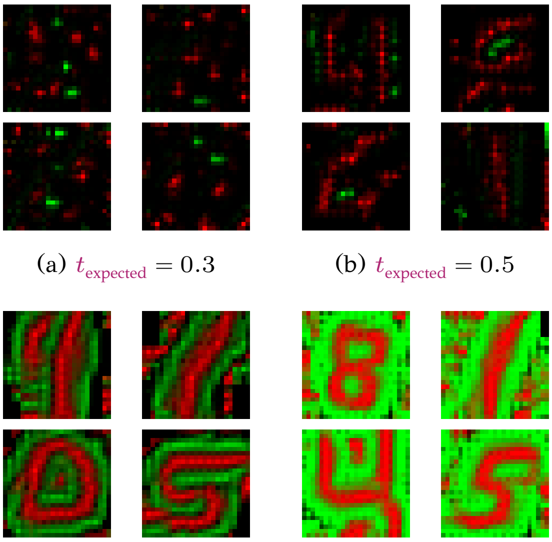
First, the impact of the parameter t_{expected} is studied. It directly impacts both the learned filters (Figure 6.2) and the classification performance (Figure 6.3). While low values of t_{expected} lead to very local patterns (Figure 6.2a), larger values lead to more global patterns (Figure 6.2c). Using late t_{expected}, and, so, training neurons to integrate a large number of spikes, helps to improve the classification rate. However, the performance decreases with very late t_{expected}: the latest spikes, which encode the lowest input values, are not useful for pattern classification. Networks with $t_{\text{expected}} = 0.75$ yield state-of-the-art results for SNNs trained with STDP on the MNIST dataset: 98.47% (see Table 6.7 for competing approaches). The two update mechanisms described in Equation 5.2 and Equation 5.3 are necessary to reach good classification

		Learning			
α	0.95	n_{epoch}	100		
		STDP			
w_{min}	0.0	w_{max}	1.0	$\eta_{\text{w}}(0)$	0.1
β	1.0	τ_{STDP}	0.1	$w(0)$	$\sim \mathcal{U}(0, 1)$
		Neural Coding			
$t_{\text{exposition}}$	1.0				
		Threshold Adaptation			
t_{expected}	0.7	$\eta_{\text{th}}(0)$	1.0	th_{min}	1.0 mV
$v_{\text{th}}(0)$	$\sim \mathcal{G}(5, 1)$ mV	v_{inh}	1.0 mV		
		Pre-processing			
$\text{DoG}_{\text{center}}$	1.0	$\text{DoG}_{\text{surround}}$	4.0	DoG_{size}	7

Table 6.1: Default SNN parameters used in the experiments. $\mathcal{G}(\mu, \sigma)$ is a normal distribution centered in μ and with variance of σ. $\mathcal{U}(a, b)$ is a uniform distribution in $[a, b]$.

Type	$\text{h}_{\text{width}} \times \text{h}_{\text{height}}$	$\lvert\mathcal{F}\rvert$	l_{stride}	l_{pad}
Convolution	5×5	32	1	0
Pooling	2×2	32	2	0
Convolution	5×5	128	1	0
Pooling	2×2	128	2	0
Fully-connected	4×4	4096	1	0

Table 6.2: Architecture used with the MNIST dataset.



(a) $t_{\text{expected}} = 0.3$

(b) $t_{\text{expected}} = 0.5$

(c) $t_{\text{expected}} = 0.7$

(d) $t_{\text{expected}} = 0.9$

Figure 6.2: Filters learned w.r.t. t_{expected} with multiplicative STDP.

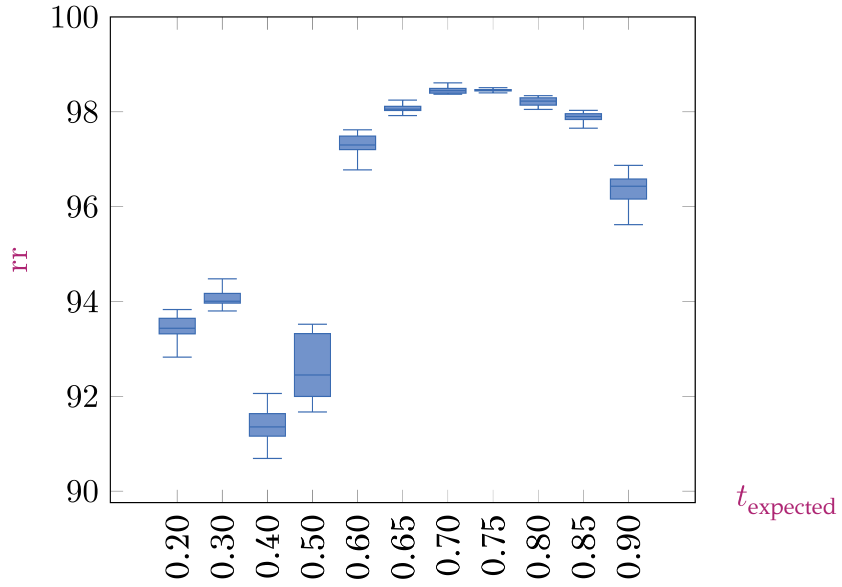


Figure 6.3: Recognition rates according to the t_{expected} parameter with biological STDP ($\tau_{\text{STDP}} = 0.1$).

rates. When Equation 5.3 is disabled in the threshold update, the homeostasis of the system is not maintained, which leads to a classification rate of $94.54 \pm 1.16\%$ when $t_{\text{expected}} = 0.75$. When Equation 5.2 is disabled, controlling the type of pattern learned becomes difficult and highly dependent on the initial values of the thresholds $v_{\text{th}}(0)$.

Using different t_{expected} values across the layers decreases the performance (Table 6.3). Let Δ_t be the difference between the t_{expected} parameters of two consecutive layers. Since neurons of the previous layer are trained to fire at specific timestamps, setting an earlier t_{expected} (i.e. $\Delta_t < 0$) on the current layer results in missing spikes from the previous neurons. Setting a later t_{expected} (i.e. $\Delta_t > 0$) results in taking into account spikes that come too late after the t_{expected} of the previous layer. A spike which arises too late compared to t_{expected} means that the current pattern is not similar to those usually recognized by the input neuron. With small values of $|\Delta_t|$, the performance of the network remains stable, which shows that the threshold adaptation mechanism is noise-resistant to some extent. However large values for $|\Delta_t|$ have a negative impact on the classification rate, especially when $\Delta_t < 0$. Δ_t inversely proportional to the sparsity: positive values of Δ_t tend to let neurons integrate more spikes and, so, allow more neurons to fire, which decreases sparsity. For $\Delta_t = -0.20$, the classification rate and sparsity are very low because the network cannot generates any spike: the t_{expected} of the second layer is defined at a timestamp where no spikes have been generated yet by the first layer. $\Delta_t = 0.01$ yields the best result: 99.53%. This small offset seems to reinforce the resistance to the noise, without integrating spikes generated by unrelated patterns. These results show that finding a single value for t_{expected} is sufficient in the exhaustive search, and the others t_{expected} can be defined by using a very small or null Δ_t. This makes it easy to set the threshold adaptation of a multilayer SNN.

Inhibition

Experiments are run in order to show the impact of the inhibition strategy on recognition rates. The three inhibition policies detailed in Section 6.2.1 are compared. Table 6.4 shows that increasing the hardness of inhibition during inference tends to decrease the recognition rate. This can be related to the sparsity level. The

Δ_t	rr	sp
-0.20	11.35 ± 00.00	0.0000 ± 0.0000
-0.10	85.56 ± 2.28	0.5129 ± 0.0230
-0.05	97.68 ± 0.14	0.2855 ± 0.0067
-0.01	98.36 ± 0.05	0.1568 ± 0.0068
0.0	98.47 ± 0.07	0.1365 ± 0.0052
+0.01	98.54 ± 0.10	0.1209 ± 0.0066
+0.05	98.43 ± 0.10	0.0754 ± 0.0082
+0.10	97.24 ± 0.24	0.0176 ± 0.0010
+0.20	92.43 ± 1.70	0.0004 ± 0.0016

Table 6.3: Result with different t_{expected} variations. Δ_t is the difference of t_{expected} between consecutive layers. T_{expected} of the first layer is fixed to 0.75.

Layer	rr	sp
	Inhibition policy	
Conv1	84.28 ± 0.98	0.3389 ± 0.0148
Conv2	89.07 ± 0.74	0.6509 ± 0.0026
FC	61.82 ± 1.92	1.0000 ± 0.0000
	Soft inhibition	
Conv1	85.47 ± 0.99	0.2806 ± 0.0443
Conv2	96.14 ± 0.68	0.3984 ± 0.0171
FC	94.86 ± 0.17	0.8965 ± 0.0031
	No inhibition	
Conv1	84.71 ± 1.04	0.1538 ± 0.0069
Conv2	96.15 ± 0.17	0.1621 ± 0.0056
FC	98.47 ± 0.07	0.1365 ± 0.0052

Table 6.4: Recognition rates with the different inhibition policies for $t_{\text{expected}} = 0.75$ and biological STDP ($\tau_{\text{STDP}} = 0.1$).

STDP rule	rr	sp
Additive STDP	96.10 ± 0.33	0.8057 ± 0.0127
Multiplicative STDP ($\beta = 2.0$)	97.99 ± 0.10	0.6298 ± 0.0052
Multiplicative STDP ($\beta = 3.0$)	98.22 ± 0.06	0.3215 ± 0.0154
Multiplicative STDP ($\beta = 4.0$)	97.67 ± 0.11	0.1203 ± 0.0044
Biological STDP ($\tau_{\text{STDP}} = 0.05$)	98.04 ± 0.14	0.0622 ± 0.0072
Biological STDP ($\tau_{\text{STDP}} = 0.1$)	98.47 ± 0.07	0.1335 ± 0.0066
Biological STDP ($\tau_{\text{STDP}} = 0.5$)	98.16 ± 0.13	0.2220 ± 0.0096

Table 6.5: Recognition rate w.r.t. STDP rules ($t_{\text{expected}} = 0.75$).

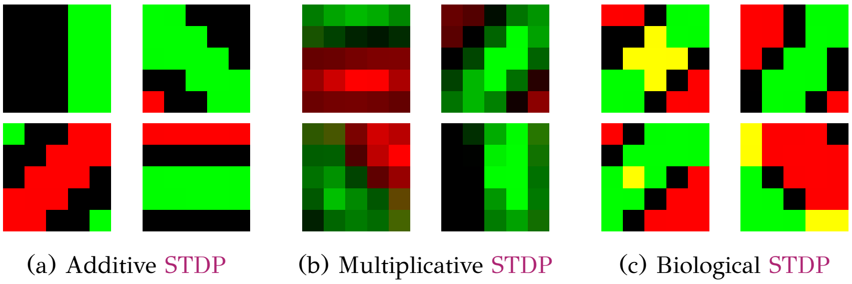
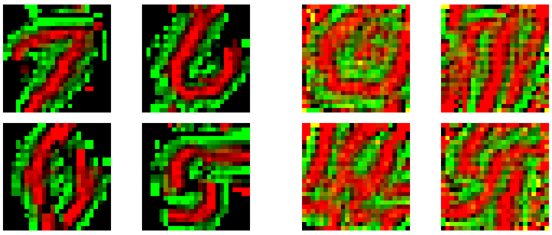

(a) Additive STDP (b) Multiplicative STDP (c) Biological STDP

Figure 6.4: Filters learned in the first convolution w.r.t. STDP.

effect of inhibition, which is minimal in the first layer, is accentuated after each layer. This effect strongly impacts both the sparsity and the recognition rate in the fully connected layer. This effect is visible with soft inhibition, but is maximal with the WTA policy: the sparsity of the fully-connected layer is 1, while the recognition rate is only 63.43%. Maintaining higher levels of activity helps to learn better representations.

STDP Rule

The effects of the STDP rules on the network classification rates and sparsity are also tested, by using the three STDP rules described in Section 2.3.4: additive STDP, multiplicative STDP and biological STDP (Table 6.5). Additive STDP gives a baseline performance of 96.10% and a relatively high level of sparsity (0.8057). Figure 6.4a shows that this STDP leads to binary weights (0 or 1) due to a saturation effect. Multiplicative STDP reduces this effect using the β parameter: large values of β reduce drastically the number of weights close to 0 or 1 (Figure 6.4b). Table 6.5 shows that increasing β decreases the sparsity. $\beta = 3.0$ gives a classification rate of 99.22% and a sparsity of 0.3215. Finally, the best performance is reached with biological STDP with $\tau_{\text{STDP}} = 0.1$ (98.47%). Decreasing this parameter also reduces the sparsity. Figure 6.4c shows that filters learned by biological STDP look different from the ones learned by other STDP rules. Indeed, additive and multiplicative STDP rules never learn patterns that overlap on the on and off channels (i.e. red and green pixels are always separated in the filters), because the input coding used does not allow to generate a spike from both channels at the same position. In contrast, biological STDP leads to filters with reinforced connections on the two channels (yellow pixels), which means that biological STDP is able to combine multiple patterns. Whatever the STDP rule, multiplicative or biological STDP, networks with the lowest levels of sparsity do not yield the best classification performances. The shapes of the fully-connected layer filters also differ between the STDP rules. While additive and multiplicative STDPs lead to easily identifiable digits (Figure 6.5a), biological STDP filters appear to be less clear



(a) Additive STDP (b) Biological STDP

Figure 6.5: Filter reconstructions of units of fully connected layers learned with different STDP rules.

$\|\mathbf{a_i}\|$	$\mathbf{t}_{\text{expected}}$	rr
4096	0.750	98.47 ± 0.07
2048	0.300, 0.750	98.51 ± 0.06
2048	0.650, 0.750,	98.53 ± 0.06
1024	0.300, 0.500, 0.700, 0.800	98.59 ± 0.06
1024	0.650, 0.700, 0.750, 0.800	**98.60 ± 0.08**
512	0.200, 0.300, 0.400, 0.500, 0.600, 0.700, 0.800, 0.900	98.48 ± 0.05
512	0.675, 0.700, 0.725, 0.750, 0.775, 0.800, 0.825, 0.850	98.57 ± 0.08

Table 6.6: Recognition rates with multiple groups, each of them containing N neurons with the same t_{expected}. Each configuration has a total of 4096 output neurons.

(Figure 6.5b). It seems that the non-linearity brought by the biological STDP allows learning more complex features, which improve performances.

Multiple Target Timestamp Networks

Finally, networks that contain several groups of neurons with different t_{expected} are investigated. Representations learned with different target timestamps can contain more diverse patterns, which can help the classifier. To do so, n independent networks are trained, where all neurons are set with a given t_{expected} value. Then, the output features of each group a are merged by concatenating them into the feature vector $\mathbf{g}$. To make a fair comparison, each configuration produce a feature vector of the same size ($|\mathbf{g}| = \sum_{i=0}^{n} |a_i| = 4096$). Table 6.6 shows that using multiple targets improves the classification performance. The network reaches a recognition rate of 98.60%, which is better than existing comparable methods (Table 6.7). One explanation can be that the combination of different t_{expected} allows detecting more varied patterns.

Model	Description	rr
Querlioz et al. 2011[111]	Single layer SNN	93.50
Dielh et al. 2015[112]	Single layer SNN	95.00
Tavanaei et al. 2016[150]	Convolutional SNN+SVM	98.36
Kheradpisheh et al. 2018[151]	Convolutional SNN+SVM	98.40
This work	Convolutional SNN+SVM	**98.60**

Table 6.7: Comparison of recognition rates of different spiking models with STDP from the literature.

Type	$h_{width} \times h_{height}$	$\|\mathcal{F}\|$	l_{stride}	l_{pad}
Convolution	5×5	32	1	2
Pooling	7×7	32	6	3
Convolution	17×17	64	1	8
Pooling	5×5	64	5	2
Convolution	5×5	128	1	2

Table 6.8: Architecture used on Faces/Motorbikes.

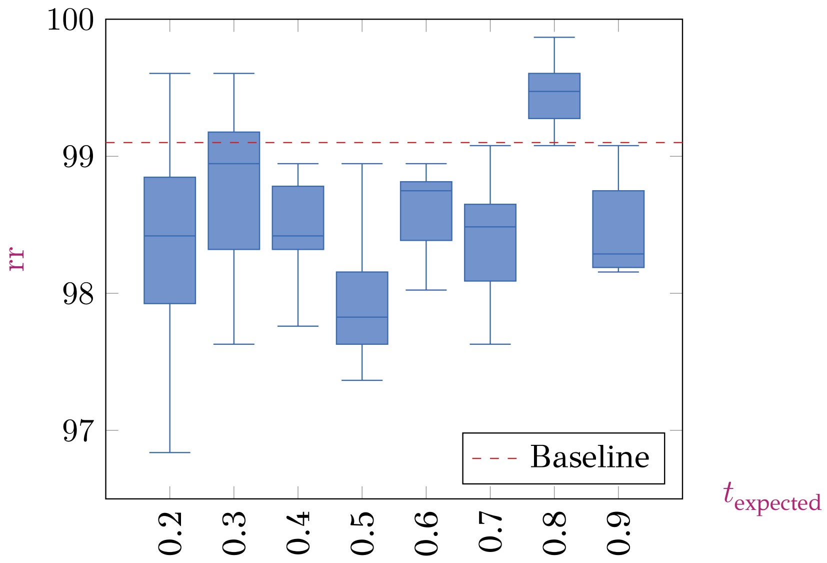


Figure 6.6: Recognition rates on Faces/Motorbikes according to the $t_{expected}$ used. The baseline is the best result reported in [151].

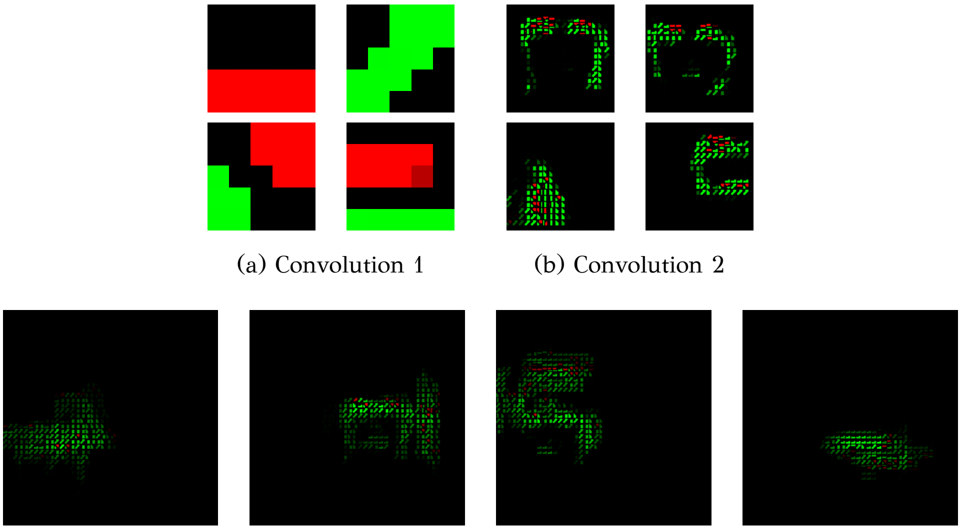
6.3.3 Faces/Motorbikes

Finally, the model introduced in this chapter is also tests on the Faces/Motorbikes dataset used in [151], in order to ensure that it also performs well on more realistic images. The dataset contains two classes extracted from the Caltech-101 dataset: faces and motorbikes. Similarly to [151], images are resized to 250×160 pixels, then converted into the grayscale format. The training set has 474 samples and the test set has 759 samples. Since the training protocol introduced in this chapter differs from [151] (Section 6.2.3), it is necessary to increase the number of filters in the convolution layer and to use larger values for th_{min} (in the following experiment, $th_{min} = 8$) to focus on patterns resulting from enough spikes. Additive STDP is used in all the convolution layers. The detailed architecture is provided in Table 6.8.

This model gives results similar to those reported in [151] (Figure 6.6), where the best reported result is 99.1%. When using $t_{expected} = 0.8$, the model performs better with an average of 99.46%. The learned filters are similar to [151] (Figure 6.7).

6.4 Discussion

The model introduced in this chapter is almost fully local and is unsupervised from the input data to the input of the classifier. However, convolutions remain an issue for implementing multilayered SNNs. Convolution columns are trained independently from the others, but it is still necessary to copy the weight and



(a) Convolution 1

(b) Convolution 2

(c) Convolution 3

Figure 6.7: Reconstruction of the receptive fields of filters learned on Faces/Motorbikes in the different layers.

threshold values to the other columns after training to mimics the weight sharing mechanism. This is required to reconstruct the geometry of the feature maps, for instance to apply pooling. In this work, a linear SVM is used to classify the output of the network, in order to be able to compare this work to the literature [150], [151]. However, in order to have a fully hardware-implementable SNN, using bio-inspired classifiers seems to be inescapable. A recent work has succeeded in using reward STDP, which is a form of reinforcement learning. This rule allows to make a multilayered SNNs that includes a spiking classifier [183]. The performance of this model with such learning rules should be investigated, while respecting the locality constraint of the computations. Finally, results showed that t_{expected} is a parameter that has a strong impact on the classification performance of the network. An interesting feature could be to introduce an auto-adaptable version of this parameter, so that neurons can find by themselves the best timing for firing. Such mechanisms would have the advantage of setting an optimal t_{expected} value for each feature independently.

6.5 Conclusion

Previous multilayered SNN models require a particular attention in setting neuron thresholds, needing an exhaustive search to be optimized. Moreover, the optimal values vary from one layer to another [151]. The threshold adaptation mechanism studied in this chapter relies on a single parameter for all the layers and allows to learn varied patterns. Experiments showed that this model leads to state-of-the-art results with unsupervised SNNs followed by an SVM on MNIST (98.60%) and on Faces/Motorbikes (99.46%). Removing the inhibition during the inference step helps to reduce the sparsity of the model activity, which leads to an improvement of the performance. Finally, studies of the impact of STDP rules showed that biological STDP helps to improve the network performance by introducing non-linearities.

Chapter 7

Conclusion and future work

7.1 Conclusion

One of the main limitations of spiking neural networks (SNNs) is their poor performances compared to artificial neural networks (ANNs), and notably deep learning. This gap does not allow SNNs to process complex data, and so, to be used in some computer visions applications. The motivation of this manuscript is to study the spiking models in order to improve the performance of image classification tasks. The solution must be as compatible as possible with neuromorphic hardware, in order to take advantage of their energy efficiency. Thus, this manuscript focuses on the spike-timing-dependent plasticity (STDP) rule, whose hardware implementation has been extensively studied [28], [197], [198]. One avenue to reach this goal is the study of multi-layered networks, which have proven their effectiveness with deep learning. However, only little work succeeds in setting up multi-layered SNN [150], [151].

The first contribution detailed in this manuscript is the development of SNN simulators. Since the creation of hardware is a laborious and expensive process, usage of software simulator is an interesting alternative to explore these models. The first simulator presented is the neural network scalable spiking simulator (N2S3), which is designed to be flexible and thus, is able to run a wide range of models (see Section 3.1). As SNNs are still immature, it is interesting to be able to quickly modify the different elements in these networks, such as the learning methods, the neuron models, or the neural coding. This flexibility is exhibited on a case study, by using three different approaches with different training methods on a motion detection task. This tool is also intended to be scalable, thanks to the actor paradigm used in the core of the simulator. However, N2S3 is not yet very effective for simulating large networks. One of the main reasons is the synchronization bottleneck, which currently requires that all timed events (e.g. spikes) go through a global synchronizer. A second simulator, the convolutional spiking neural network simulator (CSNNS), is intended to simulate a reduced number of models (e.g. integrate-and-fire (IF) neurons and temporal coding) in a very efficient way (see Section 3.2). Multi-layered SNNs tends to be large networks, which leads to long computations times. Optimizing the simulation is critical to study these networks, in order to be able to run them in a reasonable time. All the experiments reported in this manuscript are either simulated using N2S3 (Chapter 4) or with CSNNS (Chapter 5 and Chapter 6).

The next contribution focuses on the frequency loss problem, which prevents the use of several layers in SNNs (see Chapter 4). Models used in the SNN literature [111], [112] cannot be stacked because the activity across the layers drops drastically. In this chapter, three mechanisms are proposed to avoid this issue: the target frequency threshold (TFT), the binary coding, and the mirrored STDP. TFT is a threshold adaptation mechanism which trains neurons to reach a desired output frequency. Then, binary coding is a way to convert images into spike trains in order to prevent the loss of frequency. Finally, the mirrored STDP exploits the specificity of the binary coding to improve the learning speed and the network stability. Section 4.2 shows that the combination of the three mechanisms allows to maintain the frequency at the output of the network but also to keep similar recognition rates. However, binary coding has the drawback of losing information in the conversion process.

Chapter 5 proposes another threshold adaptation rule in order to allow STDP to learn patterns on samples converted with latency coding. This coding, unlike frequency and binary coding, allows to encode one continuous value with at most one spike without any information loss. In the goal of making SNNs able to process more complex data, the behavior of STDP with colored images is investigated (CIFAR-10, CIFAR-100 and SLT-10). Training SNNs directly on RGB-images gives poor filters. On/off filtering is a method that can improve the training of STDP, but is only used in the literature on greyscale images. Multiple on/off filtering policies are introduced in this chapter to test their performance on colored images. Different measures (i.e. recognition rates, activity sparsity, filter coherence, reconstruction error) are used to compare SNNs with sparse auto-encoders (AEs), a popular unsupervised ANN. Notably, it is shown that the very high sparsity induced by winner-takes-all (WTA) inhibition may lead to inefficient representations. Moreover on/off filters lead to an information loss, which also decreases the performance. However, Section 5.5.5 explores the replacement of on/off filters with the whitening transformation, which does not retain only a specific frequency. The preliminary results show the potential of this method, which reaches 66.58% on CIFAR-10.

Finally, Chapter 6 succeeds in setting up multi-layered SNNs trained with STDP. The previous threshold adaptation rule is extended in order to better control this mechanism. In addition, a protocol to train multi-layered SNNs is provided. Different mechanisms are investigated, such as the STDP rule, the inhibition system, or the threshold adaptation. This chapter shows that all the thresholds can be optimized with a single hyperparameter (t_{expected}), and that this hyperparameter allows to control the type of learned patterns. Moreover, biological STDP improves the performance compared to additive and multiplicative STDP rules, probably due to its additional non-linearity. Finally, removing the inhibition after training allows to reduce the sparsity and to increase the recognition rate. This study allows to improve the state-of-the-art results on both MNIST (98.60%) and Face/Motorbikes (99.46%) datasets.

7.2 Future Work

Some points still need to be addressed for the purpose of producing energy-efficient neuromorphic architectures able to compete with traditional methods. A first avenue lays in the improvement of the simulation of SNNs (Section 7.2.1). Indeed, large networks (e.g. deep learning), are required to process state-of-the-art datasets. Building software simulators able to run efficiently very large SNNs is an important goal in order to improve SNNs. However, the creation of scalable simulators is challenging, and require more studies. A second avenue is the enhancement of the performance of the spiking models (Section 7.2.2). The work begun in Chapter 5 should be continued, in order to address more complex datasets, such as ImageNet. Different mechanisms still need to be studied in order to better master STDP learning. A third avenue would aim to make the models described in this manuscript fully compatible with hardware implementations (Section 7.2.3). To this end, all the mechanisms used in the model should be in the spike domain and limited to local computation and memory. The final goal of this avenue would be to realize a working hardware demonstrator.

7.2.1 Simulation of Large Scale Spiking Neural Networks

Even if recent studies improved the performances of SNNs on image classification tasks, the gap between ANNs and SNNs remains huge. It seems important to be able to test the ability of STDP on state-of-the-art vision datasets, such as ImageNet, MS-COCO, or OpenImage (see Section 2.1.4). Processing these datasets may require an increase in the size of networks, just as deep learning has taken advantage of the addition of layers to surpass the state-of-the-art results. However, simulating large SNNs on software is not easy at the moment because current software simulators are not designed to do very large simulations in a reasonable time.

In order to optimize a SNN simulator to run large scale multi-layered networks, the computations must be parallelized. There are two ways to do this according to the Flynn taxonomy [199]: by applying the same operation on multiple data (single instruction multiple data (SIMD)) or by simultaneously executing different operations (multiple instruction multiple data (MIMD)). ANN simulation tends to be parallelized in with SIMD, by the use of graphical processing units (GPUs). Some work already offers implementations of SNN models on GPU [200], or the integration of SNNs into ANN frameworks [201], so that the use of GPU backends is possible. However, it is not possible to take advantage of the spatial and temporal sparsity of SNN in SIMD architectures, since all data are updated at the same time. It should be noted that some work focuses on optimizing sparse operations on GPU [202], which can help to make SNN simulation on GPU more efficient.

The second option is the use of MIMD architectures. By using a manycore processor or a distributed system, the computation can be divided into multiple execution flows. As explained in Section 3.1, such a tool takes advantage of the sparsity of SNNs to perform the computation only when it is necessary. However, the main difficulty of this solution is the requirement of synchronization between the different parallel units. Notably, such systems require that the messages (e.g. spikes) exchanged between the different units are temporally coherent. N2S3 currently uses a global synchronizer in the network, which ensures that no causality error can arise, but requiring that all timed events pass through the synchronizer. This is the main bottleneck that makes N2S3 inefficient to simulate large networks. Studies are necessary to find more scalable systems, by offering a distributed synchronization system for example. One avenue may be the usage of independent synchronizers gathering part of the network where cycles exist, and thus, where causality errors can arise [203].

7.2.2 Improving the Learning in Spiking Neural Networks

SNNs are not yet able to compete with ANNs, but several mechanisms need to be studied further in order to improve the performance of these models. The work started in this manuscript about the study of spike frequency and sparsity should be continued. We have shown that it is both necessary to maintain sufficient activity throughout the network and that too much sparsity can lead to poor performance. More work needs to be done on inhibition systems. This system must be present enough to be able to ensure that neurons are in competition and learn different patterns, but at the same time it must let enough spikes pass through.

A second avenue, following the work done in this manuscript, is the usage of the whitening transformation (see Section 5.5.5). The preliminary results presented in this section show the benefits of using this method instead of on/off filtering, but more studies are necessary to use zero component analysis (ZCA) with multi-layered SNNs. A related study is the usage of similar transformations not only at the output of the network but also in the hidden layers. In ANNs, the use of batch normalization [204], or decorrelated batch normalization [205], improves the performances because the internal covariate shift is removed at the output of the layers. An interesting study may be to investigate the presence of a similar phenomenon in spikes, and to find mechanisms that perform a similar transformation in this case. Removing these covariate shifts if these exist can be a good solution to improve performances of STDP in the upper layers.

Finally, another solution to improve learning with STDP is the use of feedback connections. Currently, STDP learns only from feedforward connections. However, this is a heavy disadvantage compared to traditional methods, such as back-propagation (BP). These methods use the product of feedforward activations and feedback errors to update the weights. This way, weights are optimized according to a global loss function. However, as seen in Section 2.3.3, regular BP, but also the BP adapted to spikes, are not yet compatible with neuromorphic hardware, for instance, because of the alternation between forward and backward steps. It could be possible to find an intermediate system, between BP, which use a global loss function, and the current STDP rules, which use only information local to the connection. An intermediate solution can be to use a loss function in each neuron, and to back-propagate the signal to only the previous layers thanks to feedback connection. This reduces the issue of the alternation between forward and backward steps since only two successive layers need to be synchronized. Some studies are necessary to test whether this intermediate method is effective on complex data and make sure that all mechanisms use only local memory and computation.

7.2.3 Hardware Implementation of Spiking Neural Networks

Current multi-layered SNNs trained with STDP are not fully implementable on neuromorphic hardware. An important task is to ensure that these models are compatible with dedicated architectures in order to take advantage of their energy efficiency. In the models described in this manuscript, three major mechanisms are not implemented in the spike domain: the pre-processing step, the shared filters in convolution layers, and the classifier.

This manuscript mainly uses on/off filters as preprocessing, performed by the convolution of a difference of Gaussians (DoG) kernel. However, this step can be realized during the value-to-spike conversion, in order to be implemented directly on neuromorphic hardware. Studies of the behavior of ganglions cells can be a solution to find a suitable implementation of on/off filters [206]. In addition, some studies are necessary to compare the performance of the spiking version of on/off filters versus the DoG version.

Concerning the convolutions, the issue comes from the weight copy step (see Section 6.2.3). Such mechanisms are required by the pooling layer, which calls for a neighborhood of similar features to apply the reduction operation. To bypass this requirement, two choices are possible: either to remove the pooling layer or to maintain similar filters in the neighborhood. The first solution is easy to set up but quickly reduces network performances. Pooling is useful because it reduces the dimension of the data while improving the position invariance. Moreover, pooling layers tend to increase the ratio of active neurons since only one spike in the receptive field of the pooling neurons is able to trigger a fire. Maintaining a correct frequency seems essential, according to the results presented in this manuscript. In addition, sharing filters across the layers allows improvements in the positional invariance of the system [2]. However, learning the filters independently should not be an issue on natural images, since the geometry is correctly distributed over the entire image.

Finally, it is necessary to incorporate the classification directly in the spike domain. Some existing work focuess on creating spiking classifiers. One of the most straightforward ways is to use supervised STDP, which uses a teacher signal to force neuron to learn desired patterns [207]. By forcing output neurons to fire when they have to (i.e. when the class associated with the output neuron is presented), the STDP rule will reinforce activated connections. It is also possible to apply anti-STDP [208] on other neurons to perform the reverse procedure, and, so, to prevent them firing when they should not. However, no studies exist yet that show that such a classifier is able to process complex data. Reward-STDP [183] is also a candidate to implement a spiking classifier. The idea is to modulate each synaptic weight update according to a reward factor. However, as for BP, some work is needed to find a solution to propagate the error signal with spikes. Some studies are required to investigate the performances of the different solutions and their ease of implementation, or to find a new one if these solutions are not satisfying.

Appendix A

Code Examples

A.1 N2S3

Listing A.1: Example of an experiment on N2S3 without domain specific language (DSL).

```
implicit val timeout = Config.longTimeout

// Creation of the simulator
val n2s3 = new N2S3("N2S3")

// Definition of the input pipeline
val inputStream = InputMnist.Entry >>
  SampleToSpikeTrainConverter[Float, InputSample2D
  [Float]](0, 22, 150 MilliSecond, 350 MilliSecond) >>
  N2S3Entry

// Load datasets
val dataFile = N2S3ResourceManager
  .getByName("mnist-train-images").getAbsolutePath
val labelFile = N2S3ResourceManager
  .getByName("mnist-train-labels").getAbsolutePath
val dataTestFile = N2S3ResourceManager
  .getByName("mnist-test-images").getAbsolutePath
val labelTestFile = N2S3ResourceManager
  .getByName("mnist-test-labels").getAbsolutePath

// Creation of the network
val inputLayer = n2s3.createInput(inputStream)
val unsupervisedLayer = n2s3.createNeuronGroup()
  .setIdentifier("Layer1")
  .setNumberOfNeurons(30)
  .setNeuronModel(LIF, Seq(
    (MembranePotentialThreshold, 35 millivolts)))

inputLayer.connectTo(unsupervisedLayer,
  new FullConnection(() => new SimplifiedSTDP))
unsupervisedLayer.connectTo(unsupervisedLayer,
  new FullConnection(() => new InhibitorySynapse))

n2s3.create()

// Add visualizations
val connectionsIndex = new ConnectionIndex(inputLayer, unsupervisedLayer)

n2s3.addNetworkObserver(new SynapticWeightSelectGraphRef(
  for(outputI <- 0 until unsupervisedLayer.shape.getNumberOfPoints) yield {
    for(inputX <- 0 until InputMnist.shape.dimensions(0)) yield {
      for(inputY <- 0 until InputMnist.shape.dimensions(1)) yield {
        val input = inputLayer.getNeuronPathAt(inputX, inputY)
        val output = unsupervisedLayer.getNeuronPathAt(outputI)
        connectionsIndex.getConnectionsBetween(input, output).head
      }
    }
  },
  SynapticWeightSelectGraph.heatMap, 4, 100))

// Run training
println("Start Training ...")
inputStream.append(InputMnist.DataFrom(dataFile, labelFile))
n2s3.runAndWait()

// Run testing
println("Start Testing ...")
unsupervisedLayer.fixNeurons()
val benchmarkMonitor = n2s3.createBenchmarkMonitor(unsupervisedLayer)

inputStream.append(InputMnist.DataFrom(dataTestFile, labelTestFile))
n2s3.runAndWait()

// Export results
println(benchmarkMonitor.getResult.evaluationByMaxSpiking)
benchmarkMonitor.exportToHtmlView("test.html")

// Destroy the simulator
n2s3.destroy()
```

A.2 N2S3 DSL

Listing A.2: Example of an experiment on N2S3 with DSL.

```
// Creation of the simulator
implicit val network: N2S3SimulationDSL = N2S3SimulationDSL()

// Definition of the input pipeline
network hasInput InputMnist.Entry >>
  SampleToSpikeTrainConverter[Float, InputSample2D[Float]]
    (0, 23, 150 MilliSecond, 350 MilliSecond) >>
  N2S3Entry

// Load dataset
val dataFile = N2S3ResourceManager
  .getByName("mnist-train-images").getAbsolutePath
val labelFile = N2S3ResourceManager
  .getByName("mnist-train-labels").getAbsolutePath
val dataTestFile = N2S3ResourceManager
  .getByName("mnist-test-images").getAbsolutePath
val labelTestFile = N2S3ResourceManager
  .getByName("mnist-test-labels").getAbsolutePath

// Creation of the network
network hasInputNeuronGroup "input"
network hasNeuronGroup "l1" ofSize 30 ofModel LIF
"l1" hasParameters (MembranePotentialThreshold -> 35.millivolts)
"l1" connectsTo "l1" using FullConnection withSynapse InhibitorySynapse
"input" connectsTo "l1" using FullConnection withSynapse SimplifiedSTDP

network buildit

// Create visualizations
observeConnectionBetween("input", "l1")

// Run training
network trainOn MnistFileInputStream(dataFile, labelFile)
```

```
// Run testing
network testOn MnistFileInputStream(dataTestFile, labelTestFile)

// Destroy the simulator
network destroyit
```

A.3 CSNN Simulator

Listing A.3: Example of an experiment on CSNNS.

```
// Creation of the experiment
Experiment<OptimizedLayerByLayer> experiment(argc, argv, "mnist");

// Definition of pre-processings
experiment.add_preprocessing<process::DefaultOnOffFilter>(7, 1.0, 4.0);
experiment.add_preprocessing<process::FeatureScaling>();

// Definition of the input converter method
experiment.input<LatencyCoding>();

// Add data to training set and test set
experiment.add_train<dataset::Mnist>(
  "train-images.idx3-ubyte", "train-labels.idx1-ubyte");
experiment.add_test<dataset::Mnist>(
  "t10k-images.idx3-ubyte", "t10k-labels.idx1-ubyte");

// Construct the network and set parameters
float t_obj = 0.75;
auto& conv1 = experiment.push_layer<layer::Convolution>("conv1", 5, 5, 32);
conv1.parameter<float>("annealing").set(0.95);
conv1.parameter<float>("min_th").set(1.0);
conv1.parameter<float>("t_obj").set(t_obj);
conv1.parameter<float>("lr_th").set(1.0);
conv1.parameter<Tensor<float>>("w").distribution<Uniform>(0.0, 1.0);
conv1.parameter<Tensor<float>>("th").distribution<Gaussian>(10.0, 1.0);
conv1.parameter<STDP>("stdp").set<stdp::Biological>(0.1, 0.1);
experiment.push_layer<layer::Pooling>("pool1", 2, 2, 2, 2);
auto& conv2 = experiment.push_layer<layer::Convolution>("conv2", 5, 5, 128);
conv2.parameter<float>("annealing").set(0.95);
conv2.parameter<float>("min_th").set(1.0);
conv2.parameter<float>("t_obj").set(t_obj);
conv2.parameter<float>("lr_th").set(1.0);
conv2.parameter<Tensor<float>>("w").distribution<Uniform>(0.0, 1.0);
conv2.parameter<Tensor<float>>("th").distribution<Gaussian>(30.0, 1.0);
conv2.parameter<STDP>("stdp").set<stdp::Biological>(0.1, 0.1);
experiment.push_layer<layer::Pooling>("pool2", 2, 2, 2, 2);
auto& fc1 = experiment.push_layer<layer::Convolution>("fc1", 4, 4, 4096);
fc1.parameter<float>("annealing").set(0.95);
fc1.parameter<float>("min_th").set(1.0);
fc1.parameter<float>("t_obj").set(t_obj);
fc1.parameter<float>("lr_th").set(1.0);
fc1.parameter<Tensor<float>>("w").distribution<Uniform>(0.0, 1.0);
fc1.parameter<Tensor<float>>("th").distribution<Gaussian>(30.0, 1.0);
fc1.parameter<STDP>("stdp").set<stdp::Biological>(0.1, 0.1);

// Setting of the number of training epoch per layer
experiment.add_train_step(conv1, 100);
experiment.add_train_step(conv2, 100);
experiment.add_train_step(fc1, 100);

// Add visualizations
conv1.plot_threshold(true);
conv1.plot_reconstruction(true);
```

```
// Create outputs
auto& fc1_out = experiment.output<TimeObjectiveOutput>(fc1, t_obj);
fc1_out.add_postprocessing<process::FeatureScaling>();
fc1_out.add_analysis<analysis::Activity>();
fc1_out.add_analysis<analysis::Coherence>();
fc1_out.add_analysis<analysis::Svm>();

// Run the experiment
experiment.run(10000);
```

Listing A.4: Example of a log automatically generated with CSNNS.

```
16:10:24 4/6/2019
Random seed: mnist_27

Run start at 16:10:24 4/6/2019
Input data [28, 28, 1]
Train:
#1: Mnist(/hd-share/datasets/mnist/train-images.idx3-ubyte,
  /hd-share/datasets/mnist/train-labels.idx1-ubyte)[60000]
Test:
#1: Mnist(/hd-share/datasets/mnist/t10k-images.idx3-ubyte,
  /hd-share/datasets/mnist/t10k-labels.idx1-ubyte)[10000]

Preprocessing 1 [28, 28, 2]:
Process.DefaultOnOffFilter {
    center_dev: 1,000000
    filter_size: 7
    surround_dev: 4,000000
}

Preprocessing 2 [28, 28, 2]:
Process.FeatureScaling {
}


Input layer [28, 28, 2]
InputConverter.LatencyCoding {
}

Layer 1: conv1 [24, 24, 32]
Layer.Convolution {
    annealing: 0,950000
    filter_height: 5
    filter_number: 32
    filter_width: 5
    lr_th: 1,000000
    min_th: 1,000000
    padding_x: 0
    padding_y: 0
    stdp: STDP.Biological {
        alpha: 0,100000
        tau: 0,100000
    }
    stride_x: 1
    stride_y: 1
    t_obj: 0,750000
    th: Gaussian(mean: 8,000000, dev: 0,100000) [32]
    w: Uniform(min: 0,000000, max: 1,000000) [5, 5, 2, 32]
}

Layer 2: pool1 [12, 12, 32]
Layer.Pooling {
    filter_height: 2
    filter_number: 32
```

```
   filter_width: 2
   padding_x: 0
   padding_y: 0
   stride_x: 2
   stride_y: 2
}

Layer 3: conv2 [8, 8, 128]
Layer.Convolution {
   annealing: 0,950000
   filter_height: 5
   filter_number: 128
   filter_width: 5
   lr_th: 1,000000
   min_th: 1,000000
   padding_x: 0
   padding_y: 0
   stdp: STDP.Biological {
      alpha: 0,100000
      tau: 0,100000
   }
   stride_x: 1
   stride_y: 1
   t_obj: 0,750000
   th: Gaussian(mean: 10,000000, dev: 0,100000) [128]
   w: Uniform(min: 0,000000, max: 1,000000) [5, 5, 32, 128]
}

Layer 4: pool2 [4, 4, 128]
Layer.Pooling {
   filter_height: 2
   filter_number: 128
   filter_width: 2
   padding_x: 0
   padding_y: 0
   stride_x: 2
   stride_y: 2
}

Layer 5: fc1 [1, 1, 4096]
Layer.Convolution {
   annealing: 0,950000
   filter_height: 4
   filter_number: 4096
   filter_width: 4
   lr_th: 1,000000
   min_th: 1,000000
   padding_x: 0
   padding_y: 0
   stdp: STDP.Biological {
      alpha: 0,100000
      tau: 0,100000
   }
   stride_x: 1
   stride_y: 1
   t_obj: 0,750000
   th: Gaussian(mean: 10,000000, dev: 0,100000) [4096]
   w: Uniform(min: 0,000000, max: 1,000000) [4, 4, 128, 4096]
}


Training step:
Layer conv1 -> 100 epochs
Layer conv2 -> 100 epochs
Layer fc1 -> 100 epochs
```

```
Output 3 of fc1 [1, 1, 4096]: mnist_27-fc1
OutputConverter.TimeObjectiveOutput {
    t_obj: 0,750000
}

Output 3, Postprocess 1 [1, 1, 4096]:
Process.FeatureScaling {
}

Output 3, Analysis: 1
Analysis.Activity {
}

Output 3, Analysis: 2
Analysis.Svm {
    c: 1,000000
}

Load 60000 train samples from Mnist(
  /hd-share/datasets/mnist/train-images.idx3-ubyte,
  /hd-share/datasets/mnist/train-labels.idx1-ubyte)[60000]
Load 10000 test samples from Mnist(
  /hd-share/datasets/mnist/t10k-images.idx3-ubyte,
  /hd-share/datasets/mnist/t10k-labels.idx1-ubyte)[10000]


mnist_27-fc1, analysis Activity:
===Activity===
* train set:
Sparsity: 0.151873
Active unit: 88.8559%
Quiet: 0%
* test set:
Sparsity: 0.153138
Active unit: 88.6765%
Quiet: 0%
mnist_27-fc1, analysis Svm:
===SVM===
classification rate: 98.2% (9820/10000)

Run end at 17:50:51 4/6/2019
Duration: 1h 40m 27s
```

Appendix B

Acronyms

AE	auto-encoder. 17, 24, 28, 41–43, 46, 73, 77, 78, 80, 82–88, 91, 93, 106
AER	address-event representation. 50
ANN	artificial neural network. 13, 14, 16, 23, 24, 28, 33, 35, 37, 39–42, 61, 70, 105–108
BCM	Bienenstock-Cooper-Munro. 39
BNN	binary neural network. 70
BP	back-propagation. 24, 40–43, 46, 108, 109
CD	contrastive divergence. 25
CMOS	complementary metal-oxide semiconductor. 15, 58
CNN	Convolutional neural network. 28, 32, 34, 41, 43
CPU	central processing unit. 45, 58
CSNNS	convolutional spiking neural network simulator. 4, 6, 17, 49, 58–60, 105, 113, 114
DBN	deep belief network. 25, 40, 41
DoG	difference of Gaussians. 22, 39, 40, 76, 81, 87, 88, 90, 108
DSL	domain specific language. 52, 60, 111, 112
DVS	dynamic vision sensor. 34
EA	evolutionary algorithm. 44
EIF	exponential integrate-and-fire. 31
FF	feed-forward. 32, 50, 54, 56–58, 95
FPGA	field-programmable gate array. 14, 15, 45
FPPA	field-programmable analog array. 15
GD	gradient descent. 24, 40
GPU	graphical processing unit. 45, 58, 107
ICA	independent component analysis. 88
IF	integrate-and-fire. 30, 31, 42, 43, 49, 74, 95, 105

IoT	Internet of things. 14, 16
JVM	Java virtual machine. 50
LAT	leaky adaptive threshold. 38, 61, 62, 65–68, 75
LIF	adaptive exponential integrate-and-fire. 31
LIF	leaky integrate-and-fire. 31, 38, 40–44, 49, 61, 64, 65, 74
LSTM	long short-term memory. 34
LTD	long-term depression. 37, 39, 42, 43, 64
LTP	long-term potentiation. 37, 39, 42, 43, 55, 63, 64
MIMD	multiple instruction multiple data. 107
MLP	multi-layer perceptron. 23, 41, 43
MNIST	Modified-NIST. 3–6, 16, 17, 25, 26, 40, 42, 44, 50, 52, 59, 65, 66, 68, 79, 80, 84, 95, 97, 98, 104
MSE	mean squared error. 87
N2S3	neural network scalable spiking simulator. 3, 4, 6, 17, 49–53, 58–60, 105, 107, 111, 112
NNMF	non-negative matrix factorization. 88
NOMFET	nanoparticle organic memory field-effect transistor. 15
PCA	principal component analysis. 88, 89
PCM	phase change memory. 15
PSP	post-synaptic potential. 42
QIF	quadratic integrate-and-fire. 31
RBF	radial basis function. 42
RBM	restricted Boltzmann machine. 25, 77
RC	reservoir computing. 54, 56–58
ReLU	rectified linear unit. 41, 42
ReRAM	resistive RAM. 15
RGB	red green blue. 19, 76, 81, 82, 85, 90, 91, 106
SGD	stochastic gradient descent. 24
SIFT	scale-invariant feature transform. 22
SIMD	single instruction multiple data. 45, 58, 107
SNN	spiking neural network. 3, 4, 14–17, 19, 28, 31, 33–42, 44–46, 49, 50, 52, 56, 58, 59, 61, 65, 70, 73, 75, 77–88, 91, 93, 95–99, 102–108
STDP	spike-timing-dependent plasticity. 3–6, 16, 17, 37, 40, 42–44, 46, 54, 55, 57, 61, 63–65, 68–71, 73, 75, 76, 78, 82–84, 86–88, 91–93, 95, 97–109
STT-RAM	spin-torque transfer RAM. 15
SURF	speeded-up robust features descriptor. 22

SVM	support vector machine. 23, 44, 59, 79, 84, 85, 97, 102, 104
TFT	target frequency threshold. 62, 65–68, 70, 105
TPU	tensor processor unit. 28
URL	uniform resource locator. 50, 51
VLSI	very large-scale integration. 15
WTA	winner-takes-all. 37, 61–63, 65, 66, 70, 75, 76, 86, 87, 93, 96, 101, 106
ZCA	zero component analysis. 89, 90, 92, 107

Appendix C

List of Symbols

Generic Parameters

α	Annealing factor. 97, 98
$*$	Convolution operator. 32, 33, 39, 40
δ	Dirac function. 30, 43
η	Learning rate. 37–39, 43, 44, 54, 55, 80
γ	Update factor. 62, 65
λ	Eigenvalue. 89
$\mathcal{G}$	Normal distribution. 53, 63, 98
$\mathcal{U}$	Uniform distribution. 98
μ	Coherence measure. 86, 87
$\mathbf{I}$	Identity matrix. 89
$\mathbf{U}$	Eigenvectors matrix. 89
$\boldsymbol{\Lambda}$	Eigenvalue diagonal matrix. 89
$\boldsymbol{\Sigma}$	Covariance matrix. 89
KL	Kullback-Liebler divergence. 78
$\mathrm{pd}_{\mathrm{max}}$	Perpendicular distance. 56
pd	Perpendicular distance. 56
sp	Sparseness measure. 85, 86, 100, 101
m_{Θ}	Trajectory orientation. 56
$m_{\mathrm{max}\,\Theta}$	Maximum trajectory orientation. 56
$m_{\mathrm{min}\,\Theta}$	Minimum trajectory orientation. 56
m_{S}	Trajectory shift. 56
m_{V}	Trajectory velocity. 56
n_{epoch}	Number of epochs. 97, 98
p	Ratio of non-null eigenvector. 89–92
t	Timestamp. 28–31, 34–36, 38, 39, 43, 53, 62–64, 69, 75, 76, 96
x	Input value. 30, 32, 34–37, 41, 56, 63, 66, 74, 77, 89
y	Output value. 35, 37, 96, 97

Neuron Parameters

F_{actual}	Actual frequency. 35, 38, 39, 62, 65–68
F_{expected}	Expected frequency. 38, 39, 62, 64–68
F_{post}	Post-synaptic frequency. 39
F_{pre}	Pre-synaptic frequency. 39
Δ_F	Relative frequency difference. 66–68

Δ_{th}	Threshold update variation. 75, 76, 96
Θ_{+}	Adaptive threshold increase constant. 38, 65
Θ_{leak}	Adaptive threshold leak time constant. 38, 65
Θ	Adaptive threshold. 38
η_{th}	Threshold learning rate. 62, 65, 75, 76, 80, 97, 98
th_{min}	Minimum threshold value. 96, 98, 103
τ_{leak}	Neuron leakage factor. 31, 38, 64, 65
θ_{th}	Frequency threshold. 39
c_{m}	Membrane capacitance. 30, 31
r_{m}	Membrane resistance. 31, 38, 65
t_{ref}	Refractory duration. 30, 31, 38, 65
v_{exc}	Spike voltage. 28, 30
v_{inh}	Inhibitory spike voltage. 96, 98
v_{rest}	Membrane resting potential. 30, 31, 65, 80
v_{th}	Membrane potential threshold. 30, 31, 38, 62, 65, 75, 76, 80, 96, 98, 99
v	Membrane potential. 30, 31, 64, 74, 96
z	Input current. 30, 31, 38

Synapse Parameters

Δ_{d}	Synaptic delay update. 54, 55
Δ_{w}	Synaptic weight update. 37, 42, 43, 55, 64
β	STDP parameter that controls the saturation effect. 43, 64, 65, 80, 83, 91, 92, 98, 101
$\eta_{\text{w}+}$	Weight learning rate in LTP. 42, 43, 64, 65, 80
$\eta_{\text{w}-}$	Weight learning rate in LTD. 42, 43, 64, 65, 80
η_{w}	Weight learning rate. 37, 55, 97, 98
τ_{STDP}	STDP time constant. 37, 55, 92, 98–101
d_{max}	Maximum synaptic delay. 76, 80
d_{min}	Minimum synaptic delay. 76, 80
d	Synaptic delay. 36, 44, 54–56, 75
r_{actual}	Actual synaptic activity trace. 43, 44
r_{expected}	Expected synaptic activity trace. 43, 44
t_{LTP}	LTP window duration. 43, 64
t_{update}	Update interval duration. 62
w_{max}	Maximum synaptic weight. 36, 43, 64, 65, 80, 83, 98
w_{min}	Minimum synaptic weight. 36, 43, 64, 65, 80, 83, 98
w	Synaptic weight. 36, 38, 39, 42–44, 56, 64, 70, 77, 78, 83, 98

Neural Coding Parameters

F_{max}	Maximum frequency. 35, 62, 65
σ_{wave}	Wave timing variance. 63–65

f_{in}	Value-to-spike conversion function. 34, 63, 74
f_{out}	Spike-to-value conversion function. 34, 35, 76
n_{wave}	Number of waves generated for each sample. 64, 68
$t_{\text{exposition}}$	Exposition duration. 35, 36, 62, 63, 65, 74–76, 80, 96, 98
t_{pause}	Pause duration. 35, 36, 62–65, 74
t_{wave}	Wave duration. 63–65
t_{wave}	Wave timing. 63
x_{th}	Input value threshold. 62, 65

Network Topology

$\mathcal{F}$	Filter set. 32, 98, 103
$\mathcal{L}$	Layer set. 32
$\mathcal{N}$	Neuron set. 32, 57, 58
$\mathcal{S}$	Synapse set. 32, 57, 58
a	Sub-network. 102
h_{height}	Filter height. 32, 33, 97, 98, 103
h_{width}	Filter width. 32, 33, 97, 98, 103
h	Filter. 32
l_{depth}	Layer depth. 32, 33, 76, 95, 97
l_{height}	Layer height. 32, 33, 95, 97
l_{output}	Output layer. 62, 65–69
l_{pad}	Padding. 33, 98, 103
l_{stride}	Stride. 33, 79, 80, 98, 103
l_{width}	Layer width. 32, 33, 95, 97
l	Layer. 32, 34, 95
n_{patches}	Number of patches. 78
n	Neuron. 29, 32, 34, 55
o_{depth}	Output depth. 97
o_{height}	Output height. 79, 97
o_{width}	Output width. 79, 97
p_{height}	Patch height. 78–80, 90
p_{width}	Patch width. 78–80, 90
q	Convolution column. 95
r_{height}	Pooling y size. 79, 80
r_{width}	Pooling x size. 79, 80
s	Synapse. 29, 32

Pre-Processing Parameters

ϵ	Whitening coefficient. 89–92
$\mathbf{W}_{\text{whiten}}$	Whitening matrix. 89, 90
$\mathbf{X}_{\text{whiten}}$	Whitened image matrix. 89
$\text{DoG}_{\text{center}}$	Variance of center Gaussian. 39, 40, 80, 98
DoG_{size}	Size of DoG filter. 39, 40, 80, 90, 98
$\text{DoG}_{\text{surround}}$	Variance of surround Gaussian. 39, 40, 80, 98
DoG	Difference of Gaussians operator. 39, 40

x_{off}	Off channel value. 40, 89
x_{on}	On channel value. 40, 89

Image Classification

Φ	Model hyper-parameters. 24, 77, 78
$\hat{\rho}$	Actual sparsity. 78
κ	Normalization factor. 78, 80, 84, 86
$\mathcal{C}$	Class set. 20, 21, 79
$\mathcal{X}_{\text{test}}$	Image test set. 22–24, 79
$\mathcal{X}_{\text{train}}$	Image train set. 22–24, 78, 79, 89
$\mathcal{Y}_{\text{test}}$	Label test set. 22, 23
$\mathcal{Y}_{\text{train}}$	Label train set. 22, 23
svm_c	SVM cost parameter. 23, 97
ρ	Expected sparsity. 62, 64, 78, 80, 84, 86
$\mathbf{X}$	Image matrix. 19–21, 23, 39, 40, 77, 78, 89, 90
$\hat{\mathbf{X}}$	Image reconstruction. 77, 78
$\mathbf{b}$	Bias. 77
$\mathbf{g}$	Feature vector. 20, 21, 76–78, 85, 86, 96, 97, 102
rr	Recognition rate. 23, 56, 57, 66–69, 99–103
υ	Sparsity factor. 78, 80, 84, 86
c	Class label. 20, 21, 23, 55
f_{c}	Classification function. 20, 21, 23
f_{dec}	Decoder function. 25, 77
f_{ec}	Image classification function. 21, 24
f_{enc}	Encoder function. 24, 77
f_{e}	Feature extraction function. 20, 21, 23, 24, 77
f_{obj}	Objective function. 24, 73, 77, 78
f_{σ}	Activation function. 77
g	Feature. 20, 85, 86, 97
n_{features}	Number of features. 21, 77–88, 91
x_{depth}	Image depth. 19, 21
x_{height}	Image height. 19, 21, 77
x_{width}	Image width. 19, 21, 77

Energy Notations

e_{dynamic}	Dynamic energy. 57, 58
e_{fire}	Energy consumed by a firing event. 57
e_{spike}	Energy consumed by a spike. 57
e_{static}	Static energy. 57, 58
e_{total}	Total energy. 57, 58
p_{neuron}	Power dissipated by a neuron. 57
p_{synapse}	Power dissipated by a synapse. 57

Spike Parameters

Δ_t	Timing difference. 37, 43, 57, 64, 99, 100

$\mathcal{D}$	Fire set. 57
$\mathcal{E}$	Spike set. 30, 57, 62, 67, 68
e	Spike. 30
f_{spike}	Spike kernel. 30
t_{expected}	Expected timestamp. 75, 80, 91, 92, 96–104, 106
t_{post}	Post-synaptic timestamp. 36, 37, 42, 43, 54, 55, 64
t_{pre}	Pre-synaptic timestamp. 36, 37, 42, 43, 54, 55, 64

Bibliography

[1] J. J. DiCarlo, D. Zoccolan, and N. C. Rust, "How does the brain solve visual object recognition?" *Neuron*, vol. 73, no. 3, pp. 415–434, Feb. 2012.

[2] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015.

[3] S. W. Lee, J. P. O'Doherty, and S. Shimojo, "Neural computations mediating one-shot learning in the human brain," *PLoS biology*, vol. 13, no. 4, pp. 1–36, Apr. 2015.

[4] P. Yger, M. Stimberg, and R. Brette, "Fast learning with weak synaptic plasticit," *Journal of Neuroscience*, vol. 35, no. 39, pp. 13 351–13 362, Sep. 2015.

[5] R. Bekkerman, M. Bilenko, and J. Langford, *Scaling Up Machine Learning. Parallel and Distributed Approaches*. Cambridge University Press, Dec. 2011, p. 4.

[6] A. H. Marblestone, G. Wayne, and K. P. Kording, "Toward an integration of deep learning and neuroscience," *Frontiers in Computational Neuroscience*, vol. 10, pp. 1–41, Sep. 2016.

[7] Y. Cao, Y. Chen, and D. Khosla, "Spiking deep convolutional neural networks for energy-efficient object recognition," *International Journal of Computer Vision*, vol. 113, no. 1, pp. 54–66, May 2015.

[8] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, Apr. 2018.

[9] G. Buzsáki, K. Kaila, and M. Raichle, "Inhibition and brain work," *Neuron*, vol. 56, no. 5, pp. 771–783, Dec. 2007.

[10] C. D. Schuman, T. E. Potok, R. M. Patton, J. D. Birdwell, M. E. Dean, G. S. Rose, and J. S. Plank, "A survey of neuromorphic computing and neural networks in hardware," *CoRR*, vol. abs/1705.06963, pp. 1–88, May 2017.

[11] S. Tang, *Deep Learning Processor List*, en-US, 2018. [Online]. Available: https://github.com/chethiya/Deep-Learning-Processor-List (visited on 06/30/2018).

[12] M. M. Waldrop, "The chips are down for moore's law," *Nature News*, vol. 530, no. 7589, pp. 144–147, Feb. 2016.

[13] J. M. Shalf and R. Leland, "Computing beyond moore's law," *Computer*, vol. 48, no. 12, pp. 14–23, Dec. 2015.

[14] D. Monroe, "Neuromorphic computing gets ready for the (really) big time," *Communications of the ACM*, vol. 57, no. 6, pp. 13–15, Jun. 2014.

[15] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, Aug. 2014.

[16] U. Government, W. House, E. O. of the President, P. C. of Advisors on Science, and Technology, "Ensuring long-term us leadership in semiconductors-2017 report, influencing china, improving us business climate, moonshots for computing, bioelectronics, electric grid, weather forecasting," Tech. Rep., Jan. 2017.

[17] B. Han, A. Sengupta, and K. Roy, "On the energy benefits of spiking deep neural networks: A case study," in *International Joint Conference on Neural Networks (IJCNN)*, IEEE, Jul. 2016, pp. 971–976.

[18] A. Joubert, B. Belhadj, O. Temam, and R. Héliot, "Hardware spiking neurons design: Analog or digital?" In *International Joint Conference on Neural Networks (IJCNN)*, IEEE, Jun. 2012, pp. 1–5.

[19] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana, "The spinnaker project," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 652–665, Feb. 2014.

[20] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, and H. Wang, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, Jan. 2018.

[21] J. Park, T. Yu, S. Joshi, C. Maier, and G. Cauwenberghs, "Hierarchical address event routing for reconfigurable large-scale neuromorphic systems," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 10, pp. 2408–2422, Oct. 2017.

[22] A. Disney, J. Reynolds, C. D. Schuman, A. Klibisz, A. Young, and J. S. Plank, "DANNA: A neuromorphic software ecosystem," *Biologically Inspired Cognitive Architectures*, vol. 17, pp. 49–56, Jul. 2016.

[23] D. Neil and S.-C. Liu, "Minitaur, an event-driven fpga-based spiking network accelerator," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 12, pp. 2621–2628, Dec. 2014.

[24] R. M. Wang, C. S. Thakur, and A. van Schaik, "An FPGA-based massively parallel neuromorphic cortex simulator," *Frontiers in Neuroscience*, vol. 12, pp. 1–18, Apr. 2018.

[25] L. P. Maguire, T. M. McGinnity, B. Glackin, A. Ghani, A. Belatreche, and J. Harkin, "Challenges for large-scale implementations of spiking neural networks on fpgas," *Neurocomputing*, vol. 71, no. 1-3, pp. 13–29, Dec. 2007.

[26] J.-H. Liu, C.-Y. Wang, and Y.-Y. An, "A survey of neuromorphic vision system-biological nervous systems realized on silicon," in *International Conference on Industrial Mechatronics and Automation (ICMA)*, IEEE, May 2009, pp. 1–4.

[27] E. Chicca, F. Stefanini, C. Bartolozzi, and G. Indiveri, "Neuromorphic electronic circuits for building autonomous cognitive systems," *Proceedings of the IEEE*, vol. 102, no. 9, pp. 1367–1388, Sep. 2014.

[28] D. Querlioz, O. Bichler, P. Dollfus, and C. Gamrat, "Immunity to device variations in a spiking neural network with memristive nanodevices," *IEEE Transactions on Nanotechnology*, vol. 12, no. 3, pp. 288–295, May 2013.

[29] W. Maass, "Noise as a resource for computation and learning in networks of spiking neurons," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 860–880, May 2014.

[30] J. J. Yang, D. B. Strukov, and D. R. Stewart, "Memristive devices for computing," *Nature Nanotechnology*, vol. 8, no. 1, pp. 13–24, Jan. 2013.

[31] B. Linares-Barranco and T. Serrano-Gotarredona, "Memristance can explain spike-time-dependent-plasticity in neural synapses," *Nature precedings*, pp. 1–4, Mar. 2009.

[32] L. Chua, "Memristor-the missing circuit element," *IEEE Transactions on Circuit Theory*, vol. 18, no. 5, pp. 507–519, Sep. 1971.

[33] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, no. 7191, pp. 80–83, May 2008.

[34] A. Pantazi, S. Woźniak, T. Tuma, and E. Eleftheriou, "All-memristive neuromorphic computing with level-tuned neurons," *Nanotechnology*, vol. 27, no. 35, pp. 1–13, Jul. 2016.

[35] D. Fan, Y. Shim, A. Raghunathan, and K. Roy, "Stt-snn: A spin-transfer-torque based soft-limiting non-linear neuron for low-power artificial neural networks," *IEEE Transactions on Nanotechnology*, vol. 14, no. 6, pp. 1013–1023, Nov. 2015.

[36] E. Gale, "Tio2-based memristors and reram: Materials, mechanisms and models (a review)," *Semiconductor Science and Technology*, vol. 29, no. 10, pp. 1–10, Sep. 2014.

[37] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, Dec. 2010.

[38] J. Grollier, D. Querlioz, and M. D. Stiles, "Spintronic nanodevices for bioinspired computing," *Proceedings of the IEEE*, vol. 104, no. 10, pp. 2024–2039, Oct. 2016.

[39] F. Alibart, S. Pleutin, O. Bichler, C. Gamrat, T. Serrano-Gotarredona, B. Linares-Barranco, and D. Vuillaume, "A memristive nanoparticle/organic hybrid synapstor for neuroinspired computing," *Advanced Functional Materials*, vol. 22, no. 3, pp. 609–616, Feb. 2012.

[40] J. Zhao and Y.-B. Kim, "Circuit implementation of fitzhugh-nagumo neuron model using field programmable analog arrays," in *Midwest Symposium on Circuits and Systems*, IEEE, Aug. 2007, pp. 772–775.

[41] M. Liu, H. Yu, and W. Wang, "FPAA based on integration of CMOS and nanojunction devices for neuromorphic applications," in *International Conference on Nano-Networks*, Springer, Sep. 2008, pp. 44–48.

[42] B. McGinley, P. Rocke, F. Morgan, and J. Maher, "Reconfigurable analogue hardware evolution of adaptive spiking neural network controllers," in *Genetic and Evolutionary Computation Conference (GECCO)*, ACM, Jul. 2008, pp. 289–290.

[43] K. Meier, "A mixed-signal universal neuromorphic computing system," in *International Electron Devices Meeting (IEDM)*, IEEE, Dec. 2015, pp. 4–6.

[44] B. V. Benjamin, P. Gao, E. McQuinn, S. Choudhary, A. R. Chandrasekaran, J.-M. Bussat, R. Alvarez-Icaza, J. V. Arthur, P. A. Merolla, and K. Boahen, "Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 699–716, Apr. 2014.

[45] N. Qiao, H. Mostafa, F. Corradi, M. Osswald, F. Stefanini, D. Sumislawska, and G. Indiveri, "A reconfigurable on-line learning spiking neuromorphic processor comprising 256 neurons and 128k synapses," *Frontiers in Neuroscience*, vol. 9, pp. 1–17, Apr. 2015.

[46] G. Montúfar, R. Pascanu, K. Cho, and Y. Bengio, "On the number of linear regions of deep neural networks," in *Advances in Neural Information Processing Systems (NIPS)*, Curran Associates, Inc., Dec. 2014, pp. 2924–2932.

[47] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *International Conference on Computer Vision (ICCV)*, IEEE, Dec. 2015, pp. 1026–1034.

[48] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, Oct. 2017.

[49] G. Kumar and P. K. Bhatia, "A detailed review of feature extraction in image processing systems," in *International Conference on Advanced Computing and Communication Technologies (ACCT)*, IEEE, Feb. 2014, pp. 5–12.

[50] K. Mikolajczyk and C. Schmid, "A performance evaluation of local descriptors," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 10, pp. 1615–1630, Oct. 2005.

[51] E. Salahat and M. Qasaimeh, "Recent advances in features extraction and description algorithms: A comprehensive survey," in *International Conference on Industrial Technology (ICIT)*, IEEE, Mar. 2017, pp. 1059–1063.

[52] G.-S. Xie, X.-Y. Zhang, W. Yang, M. Xu, S. Yan, and C.-L. Liu, "LG-CNN: From local parts to global discrimination for fine-grained recognition," *Pattern Recognition*, vol. 71, pp. 118–131, Nov. 2017.

[53] F. Liu, G. Lin, and C. Shen, "CRF learning with CNN features for image segmentation," *Pattern Recognition*, vol. 48, no. 10, pp. 2983–2992, Oct. 2015.

[54] Z. Tu, W. Xie, Q. Qin, R. Poppe, R. C. Veltkamp, B. Li, and J. Yuan, "Multi-stream cnn: Learning representations based on human-related regions for action recognition," *Pattern Recognition*, vol. 79, pp. 32–43, Jul. 2018.

[55] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, pp. 1798–1828, Mar. 2013.

[56] A. Coates, A. Ng, and H. Lee, "An analysis of single-layer networks in unsupervised feature learning," in *International Conference on Artificial Intelligence and Statistics (AISTATS)*, vol. 15, PMLR, Apr. 2011, pp. 215–223.

[57] D. Wang and X. Tan, "Unsupervised feature learning with c-svddnet," *Pattern Recognition*, vol. 60, pp. 473–485, Dec. 2016.

[58] Y. Yuan, J. Wan, and Q. Wang, "Congested scene classification via efficient unsupervised feature learning and density estimation," *Pattern Recognition*, vol. 56, pp. 159–169, Aug. 2016.

[59] H. Bourlard and Y. Kamp, "Auto-association by multilayer perceptrons and singular value decomposition," *Biological Cybernetics*, vol. 59, no. 4-5, pp. 291–294, Sep. 1988.

[60] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion," *Journal of Machine Learning Research*, vol. 11, pp. 3371–3408, Dec. 2010.

[61] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Computation*, vol. 18, no. 7, pp. 1527–1554, Jul. 2006.

[62] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," in *Advances in Neural Information Processing Systems (NIPS)*, Curran Associates, Inc., Dec. 2007, pp. 153–160.

[63] A. Makhzani and B. Frey, "K-sparse autoencoders," in *International Conference on Learning Representations (ICLR)*, Apr. 2014, pp. 1–9.

[64] G. E. Hinton, "Training products of experts by minimizing contrastive," *Neural Computation*, vol. 14, no. 8, pp. 1771–1800, Aug. 2002.

[65] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[66] Q. V. Le, N. Jaitly, and G. E. Hinton, "A simple way to initialize recurrent networks of rectified linear units," *CoRR*, vol. abs/1504.00941, pp. 1–9, Apr. 2015.

[67] Y. LeCun, F. J. Huang, and L. Bottou, "Learning methods for generic object recognition with invariance to pose and lighting," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, Jul. 2004, pp. 97–104.

[68] B. Leibe and B. Schiele, "Analyzing appearance and contour based methods for object categorization," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, vol. 2, Jul. 2003, pp. 409–416.

[69] L. Fei-Fei, R. Fergus, and P. Perona, "Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories," *Computer Vision and Image Understanding*, vol. 106, no. 1, pp. 59–70, Apr. 2007.

[70] G. Griffin, A. Holub, and P. Perona, "Caltech-256 object category dataset," California Institute of Technology, Tech. Rep., Apr. 2007.

[71] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, Dec. 2015.

[72] A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Malloci, T. Duerig, and V. Ferrari, "The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale," *CoRR*, vol. abs/1811.00982, Nov. 2018.

[73] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *European Conference on Computer Vision (ECCV)*, Springer, Sep. 2014, pp. 740–755.

[74] Y. Huang, Y. Cheng, D. Chen, H. Lee, J. Ngiam, Q. V. Le, and Z. Chen, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *CoRR*, vol. abs/1811.06965, pp. 1–11, Dec. 2018.

[75] A. Krizhevsky, "Learning multiple layers of features from tiny images," University of Toronto, Tech. Rep., Apr. 2009.

[76] K. Kowsari, M. Heidarysafa, D. E. Brown, K. J. Meimandi, and L. E. Barnes, "RMDL: Random multimodel deep learning for classification," in *International Conference on Information System and Data Mining (ICISDM)*, ACM, Apr. 2018, pp. 19–28.

[77] M. Pezeshki, L. Fan, P. Brakel, A. Courville, and Y. Bengio, "Deconstructing the ladder network architecture," in *International Conference on Machine Learning (ICML)*, vol. 48, PMLR, Jun. 2016, pp. 2368–2376.

[78] T. Cooijmans, N. Ballas, C. Laurent, Ç. Gülçehre, and A. Courville, "Recurrent batch normalization," pp. 1–13, Apr. 2017.

[79] H. Zhang, Y. N. Dauphin, and T. Ma, "Fixup initialization: Residual learning without normalization," pp. 1–16, May 2019.

[80] M. Hayat, M. Bennamoun, and S. An, "Deep reconstruction models for image set classification," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 37, no. 4, pp. 713–727, Apr. 2015.

[81] D. C. Cireşan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber, "High-performance neural networks for visual object classification," Dalle Molle Institute for Artificial Intelligence, Tech. Rep., Feb. 2011.

[82] A. Mahmood, M. Bennamoun, S. An, and F. Sohel, "Resfeats: Residual network based features for image classification," in *International Conference on Image Processing (ICIP)*, IEEE, Sep. 2017, pp. 1597–1601.

[83] E. Denton, S. Gross, and R. Fergus, "Semi-supervised learning with context-conditional generative adversarial networks," *CoRR*, vol. abs/1611.06430, pp. 1–10, Nov. 2016.

[84] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *International Symposium on Computer Architecture (ISCA)*, ACM/IEEE, Jun. 2017, pp. 1–12.

[85] P. Dayan, "Levels of analysis in neural modeling," *Encyclopedia of Cognitive Science*, Jan. 2006.

[86] J. L. Krichmar, P. Coussy, and N. Dutt, "Large-scale spiking neural networks using neuromorphic hardware compatible models," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 11, no. 4, p. 36, Apr. 2015.

[87] W. Maass, "Networks of spiking neurons: The third generation of neural network models," *Neural Networks*, vol. 10, no. 9, pp. 1659–1671, Dec. 1997.

[88] E. M. Izhikevich, "Which model to use for cortical spiking neurons?" *IEEE Transactions on Neural Networks*, vol. 15, no. 5, pp. 1063–1070, Sep. 2004.

[89] A. N. Burkitt, "A review of the integrate-and-fire neuron model: I. homogeneous synaptic input," *Biological Cybernetics*, vol. 95, no. 1, pp. 1–19, Mar. 2006.

[90] E. M. Izhikevich, "Simple model of spiking neurons," *IEEE Transactions on Neural Networks*, vol. 14, no. 6, pp. 1569–1572, Nov. 2003.

[91] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *The Journal of physiology*, vol. 117, no. 4, pp. 500–544, Aug. 1952.

[92] H. Paugam-Moisy and S. Bohte, “Computing with spiking neuron networks,” *Handbook of Natural Computing*, pp. 335–376, Jul. 2012.

[93] B. Schrauwen, D. Verstraeten, and J. Van Campenhout, “An overview of reservoir computing: Theory, applications and implementations,” in *European Symposium on Artificial Neural Network (ESANN)*, Apr. 2007, pp. 471–482.

[94] A. Borst and F. E. Theunissen, “Information theory and neural coding,” *Nature Neuroscience*, vol. 2, no. 11, pp. 947–957, Nov. 1999.

[95] R. Brette, “Philosophy of the spike: Rate-based vs. spike-based theories of the brain,” *Frontiers in Systems Neuroscience*, vol. 9, pp. 1–14, Nov. 2015.

[96] R. VanRullen and S. J. Thorpe, “Rate coding versus temporal order coding: What the retinal ganglion cells tell the visual cortex,” *Neural Computation*, vol. 13, no. 6, pp. 1255–1283, Jun. 2001.

[97] J. Huxter, N. Burgess, and J. O’keefe, “Independent rate and temporal coding in hippocampal pyramidal cells,” *Nature*, vol. 425, no. 6960, pp. 828–832, Oct. 2003.

[98] E. D. Adrian, “The impulses produced by sensory nerve-endings,” *The Journal of Physiology*, vol. 61, no. 1, pp. 49–72, Apr. 1926.

[99] D. H. Hubel and T. N. Wiesel, “Receptive fields, binocular interaction and functional architecture in the cat’s visual cortex,” *The Journal of Physiology*, vol. 160, no. 1, pp. 106–154, Jan. 1962.

[100] S. Thorpe, A. Delorme, and R. Van Rullen, “Spike-based strategies for rapid processing,” *Neural Networks*, vol. 14, no. 6, pp. 715–725, Jul. 2001.

[101] P. Reinagel and R. C. Reid, “Temporal coding of visual information in the thalamus,” *Journal of Neuroscience*, vol. 20, no. 14, pp. 5392–5400, Jul. 2000.

[102] S. Thorpe and J. Gautrais, “Rank order coding,” in *Computational Neuroscience*, Springer, Jul. 1998, pp. 113–118.

[103] S. M. Bohte, H. La Poutré, and J. N. Kok, “Unsupervised clustering with spikingneurons by sparse temporal codingand multilayer rbf networks,” *IEEE Transactions on Neural Networks*, vol. 13, no. 2, pp. 426–435, Mar. 2002.

[104] G. Tkačik, J. S. Prentice, V. Balasubramanian, and E. Schneidman, “Optimal population coding by noisy spiking neurons,” *Proceedings of the National Academy of Sciences*, vol. 107, no. 32, pp. 14 419–14 424, Aug. 2010.

[105] P. Fries, “A mechanism for cognitive dynamics: Neuronal communication through neuronal coherence,” *Trends in Cognitive Sciences*, vol. 9, no. 10, pp. 474–480, Oct. 2005.

[106] N. Hakim and E. K. Vogel, “Phase-coding memories in mind,” *PLOS Biology*, vol. 16, no. 8, pp. 1–7, Aug. 2018.

[107] C. Kayser, M. A. Montemurro, N. K. Logothetis, and S. Panzeri, “Spike-phase coding boosts and stabilizes information carried by spatial and temporal spike patterns,” *Neuron*, vol. 61, no. 4, pp. 597–608, 2009.

[108] D. O. Hebb, *The organization of behavior: A neuropsychological theory*. Wiley, 1949.

[109] G.-G. Bi and M.-M. Poo, “Synaptic modifications in cultured hippocampal neurons: Dependence on spike timing, synaptic strength, and postsynaptic cell type,” *Journal of Neuroscience*, vol. 18, no. 24, pp. 10 464–10 472, Dec. 1998.

[110] K. D. Carlson, M. Richert, N. Dutt, and J. L. Krichmar, "Biologically plausible models of homeostasis and STDP: Stability and learning in spiking neural networks," in *International Joint Conference on Neural Networks (IJCNN)*, IEEE, Aug. 2013, pp. 1–8.

[111] D. Querlioz, O. Bichler, and C. Gamrat, "Simulation of a memristor-based spiking neural network immune to device variations," in *International Joint Conference on Neural Networks (IJCNN)*, IEEE, Jul. 2011, pp. 1775–1781.

[112] P. U. Diehl and M. Cook, "Unsupervised learning of digit recognition using spike-timing-dependent plasticity," *Frontiers in Computational Neuroscience*, vol. 9, pp. 1–9, Aug. 2015.

[113] A. Lazar, G. Pipa, and J. Triesch, "Fading memory and time series prediction in recurrent networks with different forms of plasticity," *Neural Networks*, vol. 20, no. 3, pp. 312–322, Apr. 2007.

[114] C. Li and Y. Li, "A spike-based model of neuronal intrinsic plasticity," *IEEE Transactions on Autonomous Mental Development*, vol. 5, no. 1, pp. 62–73, Mar. 2013.

[115] W. Zhang and P. Li, "Information-theoretic intrinsic plasticity for online unsupervised learning in spiking neural networks," *Frontiers in Neuroscience*, vol. 13, pp. 1–14, Feb. 2019.

[116] G. G. Turrigiano, "The self-tuning neuron: Synaptic scaling of excitatory synapses," *Cell*, vol. 135, no. 3, pp. 422–435, Mar. 2008.

[117] A. Lazar, G. Pipa, and J. Triesch, "SORN: A self-organizing recurrent neural network," *Frontiers in Computational Neuroscience*, vol. 3, pp. 1–9, Oct. 2009.

[118] M. C. Van Rossum, G. Q. Bi, and G. G. Turrigiano, "Stable hebbian learning from spike timing-dependent plasticity," *Journal of Neuroscience*, vol. 20, no. 23, pp. 8812–8821, Dec. 2000.

[119] E. M. Izhikevich and N. S. Desai, "Relating STDP to BCM," *Neural computation*, vol. 15, no. 7, pp. 1511–1523, Jul. 2003.

[120] R. Rosenbaum, J. Rubin, and B. Doiron, "Short term synaptic depression imposes a frequency dependent filter on synaptic information transfer," *PLoS computational biology*, vol. 8, no. 6, pp. 1–18, Jun. 2012.

[121] T. Moraitis, A. Sebastian, I. Boybat, M. Le Gallo, T. Tuma, and E. Eleftheriou, "Fatiguing stdp: Learning from spike-timing codes in the presence of rate codes," in *International Joint Conference on Neural Networks (IJCNN)*, IEEE, May 2017, pp. 1823–1830.

[122] A. Delorme, L. Perrinet, and S. J. Thorpe, "Networks of integrate-and-fire neurons using rank order coding b: Spike timing dependant plasticity and emergence of orientation selectivity," *Neurocomputing*, vol. 38, pp. 539–545, Jun. 2001.

[123] M. S. Livingstone and D. H. Hubel, "Anatomy and physiology of a color system in the primate visual cortex," *Journal of Neuroscience*, vol. 4, no. 1, pp. 309–356, Jan. 1984.

[124] P. O'Connor, D. Neil, S.-C. Liu, T. Delbruck, and M. Pfeiffer, "Real-time classification and sensor fusion with a spiking deep belief network," *Frontiers in Neuroscience*, vol. 7, pp. 1–13, Oct. 2013.

[125] S. K. Esser, R. Appuswamy, P. Merolla, J. V. Arthur, and D. S. Modha, "Backpropagation for energy-efficient neuromorphic computing," in *Advances in Neural Information Processing Systems (NIPS)*, Curran Associates, Inc., Dec. 2015, pp. 1117–1125.

[126] S. K. Esser, P. A. Merolla, J. V. Arthur, A. S. Cassidy, R. Appuswamy, A. Andreopoulos, D. J. Berg, J. L. McKinstry, T. Melano, D. R. Barch, C. di Nolfo, P. Datta, A. Amir, B. Taba, M. D. Flickner, and D. S. Modha, "Convolutional networks for fast, energy-efficient neuromorphic computing," *Proceedings of the National Academy of Sciences*, vol. 113, no. 41, pp. 11 441–11 446, Oct. 2016.

[127] P. U. Diehl, D. Neil, J. Binas, M. Cook, S.-C. Liu, and M. Pfeiffer, "Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing," in *International Joint Conference on Neural Networks (IJCNN)*, IEEE, Jul. 2015, pp. 1–8.

[128] E. Hunsberger and C. Eliasmith, "Spiking deep networks with LIF neurons," *CoRR*, vol. abs/1510.08829, pp. 1–9, Oct. 2015.

[129] B. Rueckauer, I.-A. Lungu, Y. Hu, M. Pfeiffer, and S.-C. Liu, "Conversion of continuous-valued deep networks to efficient event-driven networks for image classification," *Frontiers in Neuroscience*, vol. 11, pp. 1–12, Dec. 2017.

[130] E. Stromatias, M. Soto, T. Serrano-Gotarredona, and B. Linares-Barranco, "An event-driven classifier for spiking neural networks fed with synthetic or dynamic vision sensor data," *Frontiers in Neuroscience*, vol. 11, pp. 1–17, Jun. 2017.

[131] Y. Bengio, D.-H. Lee, J. Bornschein, T. Mesnard, and Z. Lin, "Towards biologically plausible deep learning," *CoRR*, vol. abs/1502.04156, Aug. 2016.

[132] B. Scellier and Y. Bengio, "Equilibrium propagation: Bridging the gap between energy-based models and backpropagation," *Frontiers in Computational Neuroscience*, vol. 11, pp. 1–13, May 2017.

[133] T. P. Lillicrap, D. Cownden, D. B. Tweed, and C. J. Akerman, "Random synaptic feedback weights support error backpropagation for deep learning," *Nature Communications*, vol. 7, pp. 1–10, Nov. 2016.

[134] E. O. Neftci, C. Augustine, S. Paul, and G. Detorakis, "Event-driven random back-propagation: Enabling neuromorphic deep learning machines," *Frontiers in Neuroscience*, vol. 11, pp. 1–18, Jun. 2017.

[135] J. H. Lee, T. Delbruck, and M. Pfeiffer, "Training deep spiking neural networks using backpropagation," *Frontiers in Neuroscience*, vol. 10, pp. 1–13, Nov. 2016.

[136] C. Lee, S. S. Sarwar, and K. Roy, "Enabling spike-based backpropagation in state-of-the-art deep neural network architectures," *CoRR*, vol. abs/1903.06379, pp. 1–20, Mar. 2019.

[137] A. Tavanaei and A. Maida, "BP-STDP: Approximating backpropagation using spike timing dependent plasticity," *Neurocomputing*, vol. 330, pp. 39–47, Feb. 2019.

[138] Y. Jin, W. Zhang, and P. Li, "Hybrid macro/micro level backpropagation for training deep spiking neural networks," in *Advances in Neural Information Processing Systems (NIPS)*, Curran Associates, Inc., Dec. 2018, pp. 7005–7015.

[139] H. Mostafa, "Supervised learning based on temporal coding in spiking neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, no. 7, pp. 3227–3235, Jul. 2018.

[140] Y. Wu, L. Deng, G. Li, J. Zhu, and L. Shi, "Spatio-temporal backpropagation for training high-performance spiking neural networks," *Frontiers in Neuroscience*, vol. 12, pp. 1–12, May 2018.

[141] ——, "Direct training for spiking neural networks: Faster, larger, better," in *AAAI Conference on Artificial Intelligence*, AAAI, Jan. 2019, pp. 1–10.

[142] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in Neural Information Processing Systems (NIPS)*, Curran Associates, Inc., Dec. 2015, pp. 3123–3131.

[143] P. O'Connor and M. Welling, "Deep spiking networks," *CoRR*, vol. abs/1602.08323, pp. 1–16, Nov. 2016.

[144] P. Panda and K. Roy, "Unsupervised regenerative learning of hierarchical features in spiking deep networks for object recognition," in *International Joint Conference on Neural Networks (IJCNN)*, IEEE, Jul. 2016, pp. 299–306.

[145] T. Liu, Z. Liu, F. Lin, Y. Jin, G. Quan, and W. Wen, "Mt-spike: A multilayer time-based spiking neuromorphic architecture with temporal error backpropagation," in *International Conference on Computer-Aided Design (ICCAD)*, IEEE/ACM, Nov. 2017, pp. 450–457.

[146] G. Orchard, A. Jayawant, G. K. Cohen, and N. Thakor, "Converting static image datasets to spiking neuromorphic datasets using saccades," *Frontiers in Neuroscience*, vol. 9, pp. 1–11, Nov. 2015.

[147] L. R. Iyer, Y. Chua, and H. Li, "Is neuromorphic MNIST neuromorphic? analyzing the discriminative power of neuromorphic datasets in the time domain," *CoRR*, vol. abs/1807.01013, pp. 1–23, Jul. 2018.

[148] A. Tavanaei, Z. Kirby, and A. S. Maida, "Training spiking convnets by STDP and gradient descent," in *International Joint Conference on Neural Networks (IJCNN)*, IEEE, Jul. 2018, pp. 1–8.

[149] T. Masquelier and S. J. Thorpe, "Unsupervised learning of visual features through spike timing dependent plasticity," *PLoS computational biology*, vol. 3, no. 2, pp. 247–257, Feb. 2007.

[150] A. Tavanaei and A. S. Maida, "Bio-inspired spiking convolutional neural network using layer-wise sparse coding and STDP learning," *CoRR*, vol. abs/1611.03000, Jun. 2017.

[151] S. R. Kheradpisheh, M. Ganjtabesh, S. J. Thorpe, and T. Masquelier, "STDP-based spiking deep convolutional neural networks for object recognition," *Neural Networks*, vol. 99, pp. 56–67, Mar. 2018.

[152] A. Belatreche, L. Maguire, M. McGinnity, and Q. Wu, "A method for supervised training of spiking neural networks," *Cybernetic Intelligence, Challenges and Advances*, pp. 11–17, Sep. 2003.

[153] A. Y. Saleh, H. Hameed, M. Najib, and M. Salleh, "A novel hybrid algorithm of differential evolution with evolving spiking neural network for pre-synaptic neurons optimization," *International Journal of Advances in Soft Computing and its Applications*, vol. 6, no. 1, pp. 1–16, Mar. 2014.

[154] J. D. Schaffer, "Evolving spiking neural networks: A novel growth algorithm corrects the teacher," in *IEEE Symposium on Computational Intelligence for Security and Defense Applications (CISDA)*, IEEE, May 2015, pp. 1–8.

[155] A. V. Gavrilov and K. O. Panchenko, "Methods of learning for spiking neural networks. a survey," in *International Scientific-Technical Conference on Actual Problems of Electronics Instrument Engineering (APEIE)*, IEEE, vol. 2, Oct. 2016, pp. 455–460.

[156] R. Brette, M. Rudolph, T. Carnevale, M. Hines, D. Beeman, J. M. Bower, M. Diesmann, A. Morrison, P. H. Goodman, F. C. Harris, M. Zirpe, T. Natschläger, D. Pecevski, B. Ermentrout, M. Djurfeldt, A. Lansner, O. Rochel, T. Vieville, E. Muller, A. P. Davison, S. El Boustani, and A. Destexhe, "Simulation of networks of spiking neurons: A review of tools and strategies," *Journal of Computational Neuroscience*, vol. 23, no. 3, pp. 349–398, Dec. 2007.

[157] N. T. Carnevale and M. L. Hines, *The NEURON book*. Cambridge University Press, Jan. 2006.

[158] J. M. Bower and D. Beeman, *The book of GENESIS: exploring realistic neural models with the GEneral NEural SImulation System*. Springer Science & Business Media, Nov. 1995.

[159] M.-O. Gewaltig and M. Diesmann, "Nest (neural simulation tool)," *Scholarpedia*, vol. 2, no. 4, 2007.

[160] T.-S. Chou, H. J. Kashyap, J. Xing, S. Listopad, E. L. Rounds, M. Beyeler, N. Dutt, and J. L. Krichmar, "Carlsim 4: An open source library for large scale, biologically detailed spiking neural network simulation using heterogeneous clusters," in *International Joint Conference on Neural Networks (IJCNN)*, IEEE, Jul. 2018, pp. 1–8.

[161] D. F. Goodman and R. Brette, "The brian simulator," *Frontiers in Neuroscience*, vol. 3, pp. 1–6, Sep. 2009.

[162] T. Bekolay, J. Bergstra, E. Hunsberger, T. DeWolf, T. C. Stewart, D. Rasmussen, X. Choo, A. Voelker, and C. Eliasmith, "Nengo: A python tool for building large-scale functional brain models," *Frontiers in Neuroinformatics*, vol. 7, pp. 1–13, Jan. 2014.

[163] O. Bichler, D. Roclin, C. Gamrat, and D. Querlioz, "Design exploration methodology for memristor-based spiking neuromorphic architectures with the xnet event-driven simulator," in *International Symposium on Nanoscale Architectures (NANOARCH)*, IEEE, Jul. 2013, pp. 7–12.

[164] O. Bichler, D. Briand, V. Gacoin, B. Bertelone, T. Allenet, and J. C. Thiele, "N2d2: Neural network design & deployment," Commissariat à l'Energie Atomique et aux Energies Alternatives, Tech. Rep., Jul. 2019.

[165] A. Delorme, J. Gautrais, R. Van Rullen, and S. Thorpe, "Spikenet: A simulator for modeling large networks of integrate and fire neurons," *Neurocomputing*, vol. 26, pp. 989–996, Jun. 1999.

[166] A. P. Davison, D. Brüderle, J. M. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger, "Pynn: A common interface for neuronal network simulators," *Frontiers in Neuroinformatics*, vol. 2, pp. 1–10, Jan. 2009.

[167] P. Boulet, P. Devienne, P. Falez, G. Polito, M. Shahsavari, and P. Tirilly, "N2s3, an open-source scalable spiking neuromorphic hardware simulator," Université de Lille 1, Sciences et Technologies; CRIStAL UMR 9189, Tech. Rep., Jan. 2017.

[168] M. Shahsavari, P. Falez, and P. Boulet, "Combining a volatile and nonvolatile memristor in artificial synapse to improve learning in spiking neural networks," in *International Symposium on Nanoscale Architectures (NANOARCH)*, IEEE, Jul. 2016, pp. 67–72.

[169] M. Shahsavari and P. Boulet, "Parameter exploration to improve performance of memristor-based neuromorphic architectures," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 4, no. 4, pp. 833–846, Oct. 2018.

[170] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An overview of the scala programming language," École Polytechnique Fédérale de Lausanne (EPFL), Tech. Rep., 2004, pp. 1–20.

[171] D. Wyatt, *Akka concurrency*. Artima Incorporation, 2013.

[172] D. F. Goodman and R. Brette, "Brian: A simulator for spiking neural networks in python," *Frontiers in neuroinformatics*, vol. 2, pp. 1–10, Nov. 2008.

[173] E. M. Izhikevich, "Polychronization: Computation with spikes," *Neural Computation*, vol. 18, no. 2, pp. 245–282, Feb. 2006.

[174] H. Paugam-Moisy, R. Martinez, and S. Bengio, "A supervised learning approach based on STDP and polychronization in spiking neuron networks," in *European Symposium on Artificial Neural Networks (ESANN)*, Apr. 2007, pp. 427–432.

[175] I. Sourikopoulos, S. Hedayat, C. Loyez, F. Danneville, V. Hoel, E. Mercier, and A. Cappy, "A 4-fj/spike artificial neuron in 65 nm CMOS technology," *Frontiers in Neuroscience*, vol. 11, pp. 1–14, Mar. 2017.

[176] L. R. Iyer and A. Basu, "Unsupervised learning of event-based image recordings using spike-timing-dependent plasticity," in *International Joint Conference on Neural Networks (IJCNN)*, IEEE, May 2017, pp. 1840–1846.

[177] G. Srinivasan, S. Roy, V. Raghunathan, and K. Roy, "Spike timing dependent plasticity based enhanced self-learning for efficient pattern recognition in spiking neural networks," in *International Joint Conference on Neural Networks (IJCNN)*, IEEE, May 2017, pp. 1847–1854.

[178] A. Shrestha, K. Ahmed, Y. Wang, and Q. Qiu, "Stable spike-timing dependent plasticity rule for multilayerunsupervised and supervised learning," in *International Joint Conference on Neural Networks (IJCNN)*, IEEE, May 2017, pp. 1999–2006.

[179] B. Zhao, R. Ding, S. Chen, B. Linares-Barranco, and H. Tang, "Feedforward categorizationon AER motion events using cortex-like features in a spiking neural network," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 26, no. 9, pp. 1963–1978, Oct. 2015.

[180] R. K. Mishra, S. Kim, S. J. Guzman, and P. Jonas, "Symmetric spike timing-dependent plasticity at CA3–CA3 synapses optimizes storage and recall in autoassociative networks," *Nature Communications*, vol. 7, pp. 1–11, May 2016.

[181] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Advances in Neural Information Processing Systems (NIPS)*, Curran Associates, Inc., Dec. 2016, pp. 4107–4115.

[182] S. R. Kheradpisheh, M. Ganjtabesh, and T. Masquelier, "Bio-inspired unsupervised learning of visual features leads torobust invariant object recognition," *Neurocomputing*, vol. 205, pp. 382–392, Sep. 2016.

[183] M. Mozafari, S. R. Kheradpisheh, T. Masquelier, A. Nowzari-Dalini, and M. Ganjtabesh, "First-spike-based visual categorization using reward-modulated STDP," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, no. 12, pp. 6178–6190, May 2018.

[184] S. Zhang, J. Wang, X. Tao, Y. Gong, and N. Zheng, "Constructing deep sparse coding network for image classification," *Pattern Recognition*, vol. 64, pp. 130–140, Apr. 2017.

[185] N. Jiang, W. Rong, B. Peng, Y. Nie, and Z. Xiong, "An empirical analysis of different sparse penalties for autoencoder in unsupervised feature learning," in *International Joint Conference on Neural Networks (IJCNN)*, IEEE, Jul. 2015, pp. 1–8.

[186] K. Gupta and A. Majumdar, "Learning autoencoders with low-rank weights," in *International Conference on Image Processing (ICIP)*, IEEE, Sep. 2017, pp. 3899–3903.

[187] S. Rifai, P. Vincent, X. Muller, X. Glorot, and Y. Bengio, "Contractive auto-encoders: Explicit invariance during feature extraction," in *International Conference on Machine Learning (ICML)*, PMLR, Jun. 2011, pp. 833–840.

[188] A. Tavanaei, M. Ghodrati, S. R. Kheradpisheh, T. Masquelier, and A. Maida, "Deep learning in spiking neural networks," *Neural Networks*, vol. 111, Mar. 2019.

[189] M. D. Zeiler, "ADADELTA: An adaptive learning rate method," *CoRR*, vol. abs/1212.5701, pp. 1–6, Dec. 2012.

[190] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *Symposium on Operating Systems Design and Implementation (OSDI)*, ACM, Nov. 2016, pp. 265–283.

[191] C.-C. Chang and C.-J. Lin, "LIBSVM – a library for support vector machines," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 2, no. 3, pp. 1–27, Apr. 2011.

[192] P. O. Hoyer, "Non-negative matrix factorization with sparseness constraints," *Journal of machine learning research*, vol. 5, pp. 1457–1469, Nov. 2004.

[193] C. Savin, P. Joshi, and J. Triesch, "Independent component analysis in spiking neurons," *PLOS Computational Biology*, vol. 6, no. 4, pp. 1–10, Apr. 2010.

[194] M. Gilson, T. Fukai, and A. N. Burkitt, "Spectral analysis of input spike trains by spike-timing-dependent plasticity," *PLOS Computational Biology*, vol. 8, no. 7, pp. 1–22, Jul. 2012.

[195] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, Nov. 2004.

[196] A. Yousefzadeh, T. Masquelier, T. Serrano-Gotarredona, and B. Linares-Barranco, "Hardware implementation of convolutional STDP for on-line visual feature learning," in *International Symposium on Circuits and Systems (ISCAS)*, IEEE, May 2017, pp. 1–4.

[197] A. F. Vincent, J. Larroque, N. Locatelli, N. B. Romdhane, O. Bichler, C. Gamrat, W. S. Zhao, J.-O. Klein, S. Galdin-Retailleau, and D. Querlioz, "Spin-transfer torque magnetic memory as a stochastic memristive synapse for neuromorphic systems," *IEEE transactions on biomedical circuits and systems*, vol. 9, no. 2, pp. 166–174, Apr. 2015.

[198] G. K. Chen, R. Kumar, H. E. Sumbul, P. C. Knag, and R. K. Krishnamurthy, "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*, vol. 54, no. 4, pp. 992–1002, Apr. 2019.

[199] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE transactions on computers*, vol. 100, no. 9, pp. 948–960, Sep. 1972.

[200] A. K. Fidjeland and M. P. Shanahan, "Accelerated simulation of spiking neural networks using gpus," in *International Joint Conference on Neural Networks (IJCNN)*, IEEE, Jul. 2010, pp. 1–8.

[201] M. Mozafari, M. Ganjtabesh, A. Nowzari-Dalini, and T. Masquelier, "Spyketorch: Efficient simulation of convolutional spiking neural networks with at most one spike per neuron," *CoRR*, vol. abs/1903.02440, pp. 1–17, Mar. 2019.

[202] S. Dalton, L. Olson, and N. Bell, "Optimizing sparse matrix—matrix multiplication for the GPU," *ACM Transactions on Mathematical Software (TOMS)*, vol. 41, no. 4, pp. 1–10, Oct. 2015.

[203] C. Fidge, "Logical time in distributed computing systems," *Computer*, vol. 24, no. 8, pp. 28–33, Aug. 1991.

[204] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," pp. 448–456, Jul. 2015.

[205] L. Huang, D. Yang, B. Lang, and J. Deng, "Decorrelated batch normalization," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, Jun. 2018, pp. 791–800.

[206] R. Nelson, "Visual responses of ganglion cells," in *Webvision: The Organization of the Retina and Visual System*, N. R. Kolb H Fernandez E, Ed. University of Utah Health Sciences Center, May 2001.

[207] Y. Hao, X. Huang, M. Dong, and B. Xu, "A biologically plausible supervised learning method for spiking neural networks using the symmetric STDP rule," *Neural Networks*, vol. 121, pp. 387–395, Jan. 2020.

[208] P. D. Roberts and T. K. Leen, "Anti-hebbian spike-timing-dependent plasticity and adaptive sensory processing," *Frontiers in Computational Neuroscience*, vol. 4, pp. 1–11, Dec. 2010.