Algorithm-architecture adequacy of spiking neural networks for massively parallel processing hardware

Paul Ferré

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*

Présentée et soutenue le *11/07/2018* **par :**

PAUL FERRÉ

Adéquation algorithme-architecture de réseaux de neurones à spikes pour les architectures matérielles massivement parallèles

---

JURY

LAURENT PERRINET	Chargé de Recherche	Membre du Jury
LEILA REDDY	Chargé de Recherche	Membre du Jury

École doctorale et spécialité :
CLESCO : Neurosciences, comportement et cognition
Unité de Recherche :
CNRS Centre Cerveau et Cognition (UMR 5549)
Directeur(s) de Thèse :
Simon J. THORPE et *Franck MAMALET*
Rapporteurs :
Michel PAINDAVOINE, *Christophe GARCIA*

Declaration of Authorship

I, Paul FERRÉ, declare that this thesis titled, "Algorithm-architecture adequacy of spiking neural networks for massively parallel hardware processing" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"Ni rire, ni pleurer, ni haïr, mais comprendre"

Baruch Spinoza

UNIVERSITÉ TOULOUSE III - PAUL SABATIER

Abstract

Université Toulouse III - Paul Sabatier

Neuroscience, behaviour and cognition

Doctor of Computational Neuroscience

Adéquation algorithme-architecture de réseaux de neurones à spikes pour les architectures matérielles massivement parallèles

by Paul FERRÉ

Recent advances in the last decade in the field of neural networks have made it possible to reach new milestones in machine learning. The availability of large databases on the Web as well as the improvement of parallel computing performances, notably by means of efficient GPU implementations, have enabled the learning and deployment of large neural networks able to provide current state of the art performance on a multitude of problems. These advances are brought together in the field of Deep Learning, based on a simple modelling of an artificial neuron called perceptron, and on the method of learning using error back-propagation.

Although these methods have allowed a major breakthrough in the field of machine learning, several obstacles to the possibility of industrializing these methods persist. Firstly, it is necessary to collect and label a very large amount of data in order to obtain the expected performance on a given issue. In addition to being a time-consuming step, it also makes these methods difficult to apply when little data is available. Secondly, the computational power required to carry out learning and inference with this type of neural network makes these methods costly in time, material and energy consumption.

In the computational neuroscience literature, the study of biological neurons has led to the development of models of spiking neural networks. This type of neural network, whose objective is the realistic simulation of the neural circuits of the brain, shows in fact interesting capacities to solve the problems mentioned above. Since spiking neural networks transmit information through discrete events over time, quantization of information transmitted between neurons is then possible. In addition, further research on the mechanisms of learning in biological neural networks has led to the emergence of new models of unsupervised Hebbian learning, able to extract relevant features even with limited amounts of data.

The objectives of this work are to solve the problems of machine learning described above by taking inspiration from advances in computational neuroscience, more particularly in spiking neural networks and Hebbian learning. To this end, we propose three contributions, two aimed at the study of algorithm-architecture adequacy for the parallel implementation of spiking neural networks (SNN), and a third studying the capacities of a Hebbian biological learning rule to accelerate the learning phase in a neural network.

The first contribution consists in adapting the BCVision software kernel of

Brainchip SAS for implementation on GPU architectures. BCVision is a software library based on researches dating from 1998 that use spiking neural networks to detect visual patterns in video streams. Using this library as a basis, we first realized a parallel implementation on GPU to quantify the acceleration of processing times. This implementation allowed an average acceleration of the processing times by a factor of 10. We also studied the influence of adding complex cells on temporal and detection performance. The addition of complex cells allowed an acceleration factor of the order of one thousand at the expense of detection performance.

We then developed a hierarchical detection model based on several spiking neural circuits with different levels of subsampling through complex cells. The image is first processed in a circuit equipped with complex cells with large receptive fields in order to subsample the neural activity maps and perform a coarse pattern detection. Subsequent circuits perform fine detection on the previous coarse results using of complex cells with smaller receptive fields. We have thus obtained an acceleration factor of the order of one thousand without seriously impacting the detection performance.

In our second axis of work, we compared different algorithmic strategies for spike propagation on GPUs. This comparative study focuses on three implementations adapted to different levels of sparsity in synaptic inputs and weights. The first implementation is called naïve (or dense) and computes all the correlations between the synaptic inputs and weights of a neuron layer. The second implementation subdivides the input space into subpackages of spikes so that correlations are only made on part of the input space and synaptic weights. The third implementation stores synaptic weights in memory in the form of connection lists, one list per input, concatenated according to the input spikes presented at a given time. Propagation then involves in counting the number of occurrences of each output neuron to obtain its potential. We compare these three approaches in terms of processing time according to the input and output dimensions of a neuron layer as well as the sparsity of the inputs and synaptic weights. In this way, we show that the dense approach works best when sparsity is low, while the list approach works best when sparsity is high. We also propose a computational model of processing times for each of these approaches as a function of the layer parameters as well as properties of the hardware on which the implementation has been done. We show our model is able to predict the performances of these implementations given hardware features such as the

number of Arithmetic Logic Units (ALU) for each operation, the core and memory frequencies, the memory bandwidth and the memory bus size. This model allows prospective studies of hardware implementation according to the parameters of a neural network based machine learning problem before committing development costs.

The third axis is to adapt the Hebbian rule of Spikes-Timing-Dependent-Plasticity (STDP) for the application to image data. We propose an unsupervised early visual cortex learning model that allows us to learn visual features relevant for application to recognition tasks using a much smaller number of training images than in conventional Deep Learning methods. This model includes several mechanisms to stabilize learning with STDP, such as competition (Winner-Takes-All) and heterosynaptic homeostasis. By applying the methods on four common image databases in machine learning (MNIST, ETH80, CIFAR10 and STL10), we show that our model achieves state-of-the-art classification performances using unsupervised learning of features that needs ten times less data than conventional approaches, while remaining adapted to an implementation on massively parallel architecture.

Acknowledgements

I would like to thank first Simon Thorpe as PhD director for having supervised my PhD work and welcomed me in the CNRS CerCO laboratory.

I would also like to thank Hung Do-Duy, former director of Spikenet Technology (now Brainchip SAS), who welcomed me in the company.

I want to thank also all the Spikenet / BrainChip SAS team, who accompanied me in the discovery of the business world, and with whom many moments and meals were shared in a good mood. I would like to particularly thank Franck Mamalet for the quality of his supervision, his dedication and his valuable advices. I would like to thank also Rudy Guyonneau, who first supervised me before this PhD and allowed me to start this work.

I would also like to thank the members of the laboratory, the titulars, the post-doctoral fellows, the doctoral students and the interns, who I've been able to meet over the past four years. The scientific contribution as well as the good moments they brought me are precious.

I would particularly like to thank Perrine Derachinois, my wife, who supported, helped and loved me throughout this long journey.

I also want to thank Timothée Masquelier, Rufin Van Rullen, Douglas McLelland, Jong Mo Allegraud, Saeed Reza Kheradpisheh, Jacob Martin, Milad Mozafari, Amir Yousefzadeh, Julien Clauzel, Stefan Duffner, Christophe Garcia and obviously Simon Thorpe for all their advices and scientific support.

Finally, I would like to thank all my friends, who accompanied me, motivated me and hardened me during these four years.

Contents

List of Figures

Chapter 1

Context and state-of-the-art

Deep learning methods have recently shown ground-breaking performance levels on many tasks in machine learning. However, these methods have intrinsic constraints which limits their industrialization. Such drawbacks are a direct consequence of the deep nature of the neural network used, which requires enormous amount of computations, memory resources and energy consumption. Several research projects attempt to solve this issue by finding ways to accelerate learning and inference at software and hardware levels. One of the main advances that allowed allowed Deep Learning to become efficient and popular was the development of Graphical Processing units (GPU).

On the other hand, Brainchip Inc has used spiking neural network (SNN) based technology in order to perform fast visual pattern detection. Such networks, whose first purpose is the development of more realistic biological simulations, show interesting features for energy consumption reduction. As a matter of fact, BCVision, a software library for visual pattern detection developed in 2004 (under the name SNVision back then), is able to perform one-shot learning of novel patterns and detect objects by propagating information in the form of spikes. While BCVision is not as ubiquitous as Deep Learning methods, its principles are all biologically plausible, requires a low amount of data and require fewer resources.

This thesis proposes to explore the adequation of spiking neural networks and biological priors to parallel computing devices such as GPU. In this section we present the context of our research. We first show that the success of Deep Learning methods relies on processes similar to those seen in biological brains. We present next computational neuroscience models that are relevant for explaining the energy-efficiency of the brain, and are marginally considered in the machine learning community. We also present research on GPU optimizations of both deep neural networks and spiking neural networks, as well as the GPU

programming model which will be of primary importance for the content of this thesis. We finally explore the current research trends in Deep Learning, in neural network hardware optimization and the relationship between machine learning and neuroscience.

1.1 Deep Learning methods and their links to Biology

The fundamentals of Deep Learning

In the last decade, the machine learning community adopted massively Deep Learning methods, a set of algorithms based on neural networks architectures that can involve hundreds of layers. These methods have shown outstanding performances on many tasks in computer vision (Krizhevsky et al., 2012; He et al., 2016), speech recognition (Hannun et al., 2014; Amodei et al., 2016; Zhang et al., 2017), natural language processing (Mikolov et al., 2013b,a) and reinforcement learning (Mnih et al., 2013; Gu et al., 2016; Mnih et al., 2016).

Deep learning architectures use as a basis formal neurons (McCulloch and Pitts, 1943), derived from the perceptron model (Rosenblatt, 1958), a simple unit which performs a linear combination of inputs and their synaptic weights, followed by a non-linear function in order to compute its activation. Perceptron units are organized in layers, each layer receiving as inputs the activation from its previous layer(s). A network with multiple layers of perceptrons in called a Multi-Layer Perceptron (MLP). The Universal Approximator Theorem Cybenko (1989) states that a two-layer MLP (with one hidden layer and an output layer) with a large enough number of neurons with sigmoid activation function can approximate any continuous function. It was also shown that only the non-linear behaviour of the activation function matters in order for an MLP to approximate any continuous function (Hornik, 1991).

Learning with a MLP is performed with the backpropagation algorithm in a supervised manner. The original article on perceptrons (Rosenblatt, 1958) proposed a first learning algorithm which allowed the last layer of an MLP to learn a mapping between its inputs and a reference output (called labels or targets in the literature). However, this method did not provide a way to perform learning in multi-layer architectures. The backpropagation algorithm (Linnainmaa, 1970; Werbos, 1982; Rumelhart et al., 1986; LeCun et al., 1988) allows a

multi-layer neural network to learn its internal representations from the error of prediction. Backpropagation is a gradient descent method, where for each input a feedforward pass through the neural network is performed in order to obtain a prediction. As each input is associated to a label (or target), the error of prediction is computed between the neural network prediction and the target. This error is then backpropagated through the network by the computation of each layer's gradients from top to bottom, respectively to its inputs. Gradient computations are based on the chain-rule (Dreyfus, 1962), which allows the computation of the partial derivative of composition function. In other words, for a given parameter in the network (an activation or a synaptic weight), the chain-rule can approximate the error induced by this specific parameter given its inputs values and its output gradients. The main requirements for using backpropagation are the labelling of every input in the training dataset and the differentiability of every operation in the neural network. Note that this the last condition is violated with formal neurons (Rosenblatt, 1958), where activation function is a threshold (or Heavyside) function, which is non-derivable.

The development of such neural networks methods faced several obstacles before. First, MLPs suffers a lot from the curse of dimensionality (Bellman, 1961), meaning that with more input dimensions more neurons and samples are needed in order to avoid overfitting. Also before the extensive use of GPUs (Graphical Processing Units) for Deep Learning, computation times were a significant issue, since a single training of a single feedforward neural network could take several days to weeks. As of today, such obstacles have been largely overcome as we shall see in the next section.

From MLPs to Deep Learning: biologically plausible priors

Multi-Layer Perceptrons suffer from operational drawbacks which hinder its ability to effectively approximate universally any continuous function as theorized by Cybenko (1989). We will see in this section how regularization techniques allowed artificial neural networks to overcome those drawbacks and to become the main method in machine learning. We show that many of these regularizations take inspiration from biological models of the brain, or at least have similarities to what can be found in biological neural networks.

Data availability It has been a common philosophy concern that the availability of diverse observations helps humans to gain accuracy in their perception of reality. Plato's Allegory of the Cave states the impossibility for people with constrained observation scope to perceive correctly reality. Saint Thomas d'Aquin also proposed the concept of passive intelligence, the idea that human intelligence builds itself through experience of their environment. The more observation, the more a human or an animal is able to approximate reality, then take good decisions for its survival. This idea is supported by experiments in kittens showing that directional sensitivity (Daw and Wyatt, 1976) and orientation sensitivity (Tieman and Hirsch, 1982) can be modified during critical phases of their brain development by changing the environment. The fundamental laws of statistics and the curse of dimensionality both draw to the conclusion that an insufficient amount of data leads to variance problems, hence increasing the amount of data allows theoretically better generalization.

With the presence of huge amounts of data available via the Web, it has been possible to gather and annotate large datasets. For instance the ImageNet dataset (Deng et al., 2009), a famous dataset in computer vision, contains millions of labelled images available for training. With the addition of random transformations and Drop Out (Srivastava et al., 2014) in order to artificially increase the number of samples, models are now able to generalize better. We should notice that even millions of samples are not sufficient in order to avoid the Curse of Dimensionality. Deep Learning systems may simply overfit the training dataset, but since they contain a huge number of configurations for each label, then it is able to generalize.

Convolutions Hubel and Wiesel (Hubel and Wiesel, 1962) proposed a model of hierarchical visual processing based on simple and complex cells. Simple cells are neurons with a restricted receptive field corresponding to a subregion of the field of view. Complex cells pool together visual information relative to an orientation over a patch, allowing robustness to translations, scaling and rotations of visual patterns. The HMAX model (Riesenhuber and Poggio, 1999) implements a model on these principles that attempts to simulate processing in the ventral and dorsal visual streams. In the HMAX model, V1 simple cells are sensitive to oriented bars (Poggio and Bizzi, 2004), while subsequent simple layers encodes all combination of inputs (for V2 all the possible orientation combination). Complex cells in HMAX apply a softmax operation in order to perform their pooling operation. However, this model has no ability to

learn novel visual representations in its intermediate layers, relying instead on a SVM classifier (Vapnik, 1999). The HMAX model is also slow in terms of processing times due to the number of potential combinations in all the simple layers from V2. Nevertheless, Serre et al. (2007) proposed a comparison of the HMAX model with a simple unsupervised learning scheme (synaptic weights of a randomly chosen unit are mapped to a random afferent vector) and human during an animal/non-animal classification task, and showed that such model behaviour is very closed to human's one. It has also been shown that complex cells may not perform softmax operation for pooling, but rather a simple max operation (Rousselet et al., 2003; Finn and Ferster, 2007).

Artificial neural networks using simple cells with restricted receptive fields include LeNet (LeCun et al., 1998), which performs handwritten digit classification on the MNIST dataset with high accuracy. LeNet relies on the convolution operator in order to compute neural activations of simple cells, with one neuron's receptive field being applied across all the spatial dimensions of the image. Convolutional Neural Networks (CNN) then uses weight sharing, the assumption that for a given output activity map, all the neurons associated to this activity map have the same synaptic weights over their receptive field. This assumption is wrong with respect to biology, but this approximation leads to two advantageous features for machine learning. Such constrained receptive fields allow the model to be more compact in memory, and also reduce the degree of freedom during learning since only a few synapses can be altered contrary to the fully-connected scheme typically used in MLPs. CNNs also have complex cells after each simple cell layer that performs max-pooling and subsampling neural activity over the spatial dimension further reducing both the computations and memory requirements for subsequent layers. Finally, in order to avoid runaway dynamics of the synaptic weights during learning, a weight decay (Krogh and Hertz, 1992) term is usually added to the update equation. The weight decay is a constraint on the norm of the synaptic weights. The most popular weight decay term is the $L2$-norm, and is also named Lasso in the statistics literature (Tibshirani, 1996).

Activations Non-spiking neural models (McCulloch and Pitts, 1943; Rosenblatt, 1958) have been equipped with diverse non-linearity functions like threshold, sigmoid and hyperbolic tangent (tanh) in order to approximate the rate of spiking in a non-linear regime. However, such non-linear functions, when differentiable, suffer a lot from the vanishing gradients problem when applying

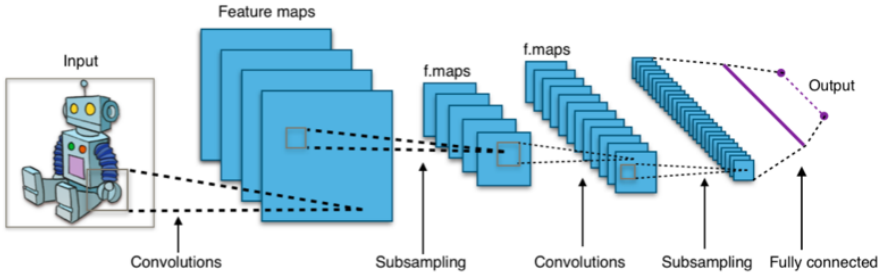


FIGURE 1.1: Standard architecture of a convolutional neural network. (Image from https://commons.wikimedia.org/ under the Creative Commons 4 license)

backpropagation rule (Hochreiter et al., 2001), resulting in infinitesimal amplitudes of weight updates for the bottom layers in a hierarchy.

A biologically plausible non-linear behaviour relying on a simple piecewise-defined linear function was proposed by Hahnloser et al. (2000). This function has been shown to correlate with physiological activity in MT cells (Rust et al., 2006) along with divisive normalization. The function assumes that if the activity of a neuron before the non-linearity is lower than zero, then the non-linear function outputs zero, else it outputs the given activity. Such activation function have been later named Rectified Linear Unit (ReLU) (Nair and Hinton, 2010) and was successfully applied in Restricted Boltzmann Machines and Convolutional Neural Networks (Krizhevsky et al., 2012). The ReLU activation function bypasses the vanishing gradient problem since its derivative for input values greater than zero is one, hence preserving the gradients amplitude.

Normalization Neural feedback inhibitions seem to play a role in contrast invariance in many sensory circuits. Carandini and Heeger (1994) showed that excitatory-inhibitory circuits may implement a divisive normalization scheme (also known as shunting inhibition) in order to limit the effect of input variation in amplitude. The presence of such context-dependent normalization have been shown to occur in V1 cortcical circuits (Reynaud et al., 2012). As we have seen in the previous paragraph, shunting inhibition with a rectifier function correlates with biological recording of MT cells (Rust et al., 2006).

In the machine learning domain, special cases of normalizations are known as whitening processes. Whitening prior to propagation has been shown to help convergence during learning with back-propagation (Wiesler et al., 2011). In order to facilitate convergence in deep CNNs, the Batch Normalization method (Ioffe and Szegedy, 2015) mean-centers and scales by the inverse variance to

apply such whitening process. In Batch Normalization, the mean and variance statistics of each neuron is learnt over all the dataset and fixed for inference phase. Hence, statistics learnt with a given dataset may not be generalizable to another dataset. This issue was addressed by the proposal of Layer Normalization (Li et al., 2016) and Instance Normalization (Huang and Belongie, 2017) where the mean and the variance are computed online (thus not learnt) over the current batch or the current sample respectively. These different normalization schemes, while coarsely approximating biological neural normalization, effectively reduce the number of iteration required for convergence.

Recurrent neural networks The Long-Short-Term-Memory (LSTM) unit (Hochreiter and Schmidhuber, 1997) is a recurrent neural model equipped with gates, inspired from the ion channels in biological networks, which is able to learn long-term dependencies in temporal data. It is often used for natural language and audio processing. The Gated Recurrent Unit (GRU) (Cho et al., 2014) is a simpler model of recurrent neuron which has been shown to perform equivalently to an LSTM unit.

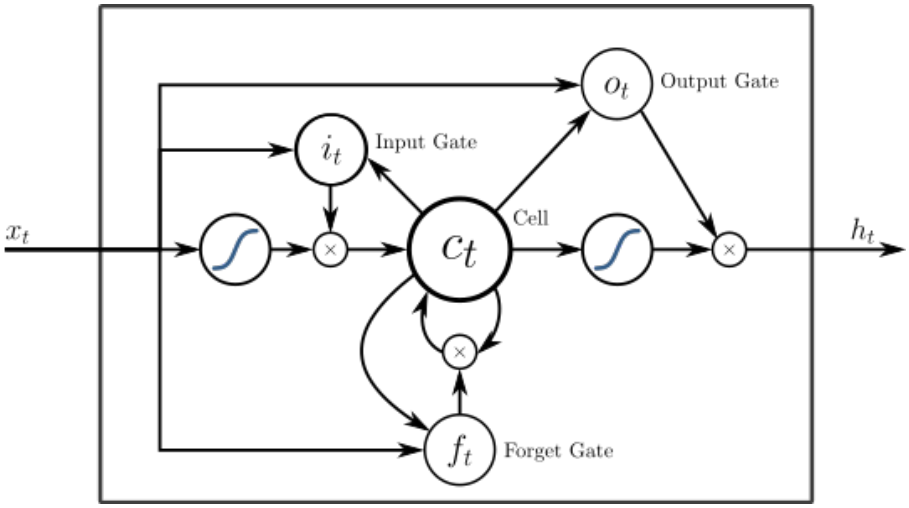


FIGURE 1.2: Architecture of an LSTM neuron (Image from https://commons.wikimedia.org/ under the Creative Commons 4 license)

1.2 Toward further biological priors for neural network acceleration

In this section we consider several biological neural mechanisms which have emerged from neuroscience research and are rarely considered in the deep learning literature.

A first mechanism is the spiking behavior of neural networks used in computational neuroscience. The complete behaviour of spiking neurons have been described by Hodgkin and Huxley (1952) using electrical stimulation of squid nerves. Spiking neurons shows temporal dynamics according to their dendritic inputs. Input spikes are modulated by synapses' conductance, and potentiation occurs at the soma. When the potential reaches a certain threshold, the neuron is depolarized and emits a spike along its axon, which is connected to other neurons dendrites. The Hodgkin-Huxley model of the neuron, while complete, is based on differential equations which computes the internal state of the neuron, thus resulting in a complex model. Simpler models are often used for simulation purposes, such as the Lapicque model (Abbott, 1999), also known as the leaky-integrate and fire (LIF) neuron. The LIF model captures the dot product behaviour between inputs and synapses, the thresold function and incorporate a leakage parameter. Izhikevich neurons (Izhikevich, 2003) also incorporate such behaviour in a very parametrizable manner and have been shown to reproduce many of the biological neural dynamics observed experimentally. The interesting factor of such neural models from a computational point of view is the nature of spikes, mathematically expressed as a Dirac distribution, or in the discrete case as a Kronecker of unit amplitude. We can assume that biological neurons output a binary information at each timestep, whether the threshold has been reached or not. This shows a fundamental difference between spiking neurons and LSTMs that encode information after the non-linearity as a floating-point real number. Spiking neurons also reset their potential right after firing, where LSTMs only reset their activity given their internal states and learnt gates weights. We can reasonably assume that spiking neurons may help in simplification of existing recurrent neural models.

Another interesting function of the brain is its ability to perform ultra-rapid visual categorization (Thorpe et al., 1996). Humans and primates are able to detect visual patterns in under 100 ms, leading to information being propagated during at most 10 ms in a single visual layer. The first wave of spikes is then probably sufficient for rapid visual categorization (Thorpe et al., 2001), since at

most one spike per neuron can occur in this 10ms frame. Rate coding is not able to explain such processing, since it cannot be measured with only one spike per neuron, thus spikes-timing coding is privileged for such task. As absolute timing differences are small, it was hypothesized through the rank order coding theory (Van Rullen et al., 1998; Thorpe et al., 2004) that a single spike per neuron is enough to perform rapid categorization of visual stimuli. In such framework, the absolute timing of spikes may be ignored, the relative order of the spikes being able to help discriminate different visual patterns. The adequacy of fully feedforward neural processing is also supported by the architectures of CNNs for computer vision tasks, where information in only propagated in a feedforward manner during inference. While CNNs typically compute neural activity as floating-points rates instead of binary spikes, we argue that rank order coding may provide useful constraints for accelerating deep learning methods applied to vision.

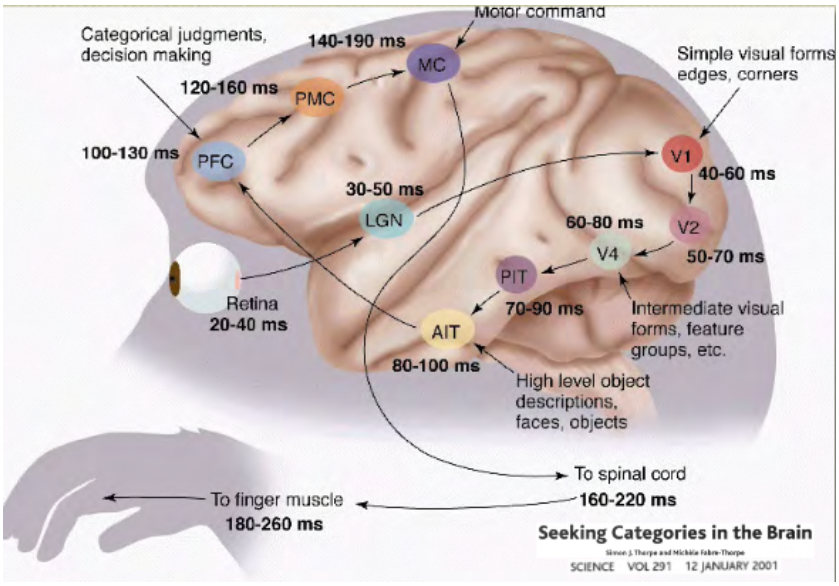


FIGURE 1.3: Latencies observed in each layer during a rapid visual categorisation task. (Image reproduced with permission from Thorpe et al. (2001))

For spiking neural networks to learn representations, the backpropagation algorithm may not be applied since spiking behaviour are the result of a non-derivable threshold based non-linearity. Further more, it is biologically implausible that supervised learning schemes such as backpropagation could be the main learning mechanisms. Many representation appear to be learned in a unsupervised way, either by determined developmental functions (Ullman et al., 2012; Gao et al., 2014), innate social learning behaviours (Skerry and Spelke, 2014; Hamlin, 2015) or physiological unsupervised mechanisms (Markram et al.,

1997; Bi and Poo, 1998). One of the main unsupervised learning mechanisms occurring in the brain is called Spike-Timing-Dependent-Plasticity (STDP). STDP is a Hebbian learning rule based on the timing differences between pre and post-synaptic spikes. In machine learning terms, STDP acts as a spike-based coincidence detector and has been shown to allow neurons to rapidly learn novel representations and capture input statistics (Delorme et al., 2001; Perrinet and Samuelides, 2002; Guyonneau et al., 2005; Masquelier and Thorpe, 2007; Masquelier, 2017). The adaptation of STDP to machine learning paradigm may serve the purposes of learning phase acceleration and bringing biological plausibility to deep networks.

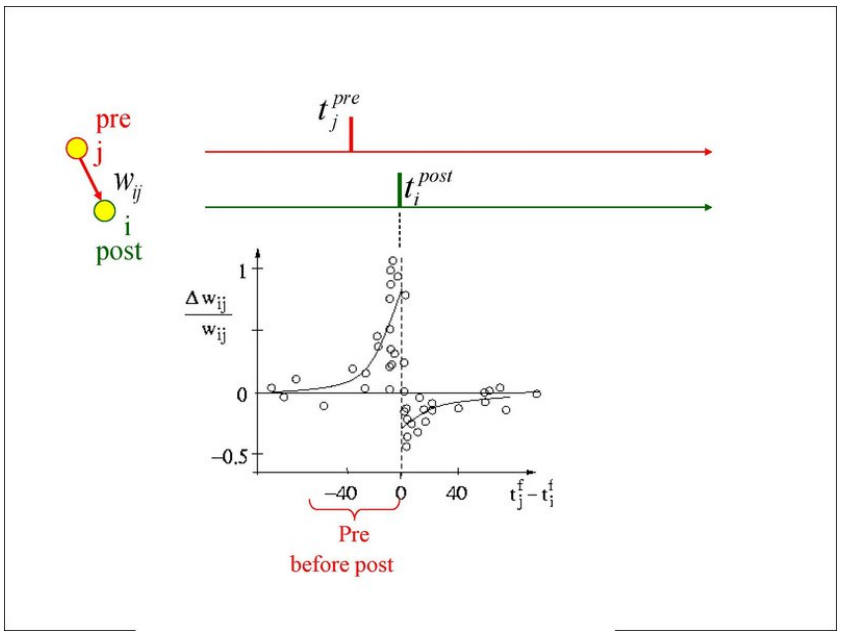


FIGURE 1.4: Synaptic changes as a function of the spike timing difference of a pre and a post-synaptic neurone ((Image from http://www.scholarpedia.org/ under the Creative Commons 3 license)

1.3 Hardware dedicated to parallel computing are suitable for neural networks acceleration

We have seen in the previous section that regularization techniques allow artificial neural network models to overcome their intrinsic issues. Regularization is generally implemented as model-based (or software-based) enhancement. In

this section we describe hardware-based enhancement brought by the algorithm-architecture adequacy of neural networks models on diverse devices, in particular on Graphical Processing Units (GPU).

GPU acceleration of deep learning methods From the machine learning perspective, the major breakthrough was the publication by Krizhevsky et al. (2012), showing that is was possible to implement a deep convolutional neural network on a GPU: the famous AlexNet. The AlexNet architectures contains seven convolutional layers for a total number of 650 thousand neurons and 630 millions synapses. Such network on CPU would take far too long to train. The shift to GPU hardware allowed a drastic acceleration of processing times, and allowed the authors to train this network on two consumer-grade NVidia GTX 580 in less than a week for a hundred epoch on the whole ImageNet dataset. This represents an acceleration of an order of magnitude of ten. Since then, GPU have been massively adopted for deep learning. The different enhancement brought by NVidia to their GPU hardware and software library CUDA as well as the massive share of open-source code between academic and industrial actors in the field led to the rapid development and improvements of many Deep Learning framework like Caffe (Jia et al., 2014), Theano (Theano Development Team, 2016), Tensorflow (Abadi et al., 2015), MXNet (Chen et al., 2015) and many others as well as convenience wrappers.

GPU acceleration of biological neural networks On the computational neuroscience side, different tools have been proposed in order to accelerate simulations of biological neural networks. Frameworks like Brian (Goodman and Brette, 2009) and CarlSim (Beyeler et al., 2015) propose a unified design architecture of neural networks which can then be run on several devices such as CPU, GPU and dedicated hardware like IBM's TrueNorth (Merolla et al., 2014) and recently Intel's LoiHi chip (Davies et al., 2018). The latter dedicated hardware pieces, while very energy-efficient, are research oriented, with a focus on biologically accurate simulations. Such simulations rely on very complex models, hence the possibility of deploying deep architectures on dedicated hardware and GPUs remain limited and does not fit operational requirements for applications. Finally, GPUs are far more accessible than dedicated hardware for both purposes, since they are basic components of computers and serve other purposes like other scientific and graphical computations for a lower price (considering consumer-grade GPUs).

CUDA programming model GPU technology has drastically improved over the last years, in terms of both computational efficiency and ease of development of such platforms. NVidia Corporation made several improvements in order to shift from their initial specialization on graphical processing toward scientific computations, and more particularly on neural network acceleration. With the addition of tensor cores (dedicated matricial computation units) on GPUs and convenient programming tools such as the CuDNN library for deep learning, Nvidia has taken the lead on the neural network accelerator market. AMD on the other hand was not as successful in this initiative, but strategical shifts in their programming language from OpenCL to HIPs, which takes inspiration from the NVidia CUDA programming language and tries to unify programming on both platforms, make it possible for previously NVidia-only projects to be run on AMD platforms. Thanks to HIPs implementation of deep learning framework, recent benchmarks have shown that the recent AMD Vega architecture may perform better than NVidia GTX Titan X GPU in deep neural networks training (GPUEater, 2018). In this part we will describe the CUDA programming model in order to highlight the constraints inherent to parallel computing, since the new HIPs framework follows the same model.

NVidia GPUs require computations to be split between several parallel computation units called Streaming-Multiprocessors (SM). Each SM contains thousands of registers partitioned dynamically across threads, several memory caches to reduce memory access latencies (which will be detailed later in this section), a warp scheduler which quickly switches context between threads and issues instructions and many dedicated execution cores for different operations at various resolutions (mainly using 32-bits integers and floating -points numbers).

SMs are designed to perform computations in a Single-Process-Multiple-Data (SPMD) manner, i.e. a single instruction is performed by the SM on each clock cycle on multiple different data points at the same time. In hardware, threads are organized in warps, a warp being a set of 32 threads performing the same computation. Instructions are issued to dedicated units, which can vary in terms of performances depending on the bit resolution and the nature of the operation. For instance, 32-bits integer additions can be performed on a single SM in four clock cycles, but the compute-units can pipeline processing and process four warps at the same time. The efficiency of an operation is given by its throughput, which is roughly the amount of data on which the operation can be performed on one clock cycle. In our example, the throughput of the integer addition is 128, since four warps can be pipelined at the same time on

the dedicated compute unit.

Data transfer between the host memory (the RAM module) and the SM is divided between several memory-cache layers.

- The global device memory is the largest but slowest memory space embedded on the GPU. As of today its capacity can reach dozens of gigabits of data. Data transfers are done through PCI-Express port (where PCI stands for Peripheral Component Interconnect), hence the data rate transfer between the host and the global memory is limited to 20 Gbits/s. Also, this memory currently relies on GDDR5 (Graphical Double Data Rate) or HBM2 (High Bandwidth Memory) technologies, on which read and write instructions from SM have longer latencies than on classical CPU architectures. From the GPU perspective, data are accessed through large memory buses (256 to 384 bits for GDDR5, 2048 to 4096 bits for HBM2) by loading large chunks of aligned and coalesced memory spaces. This is a major constraint of the parallel programming model as the violation of memory access patterns induces long latencies. Much care should also be taken regarding concurrent writes in global memory, which can result in data inconsistency. Atomic operations, i.e. writing in a memory location in a thread-safe manner, are supported but also increase latency. Such operations should not be used extensively in order to keep the advantage of parallel processing on computation times.

- When data chunks have to be accessed in a read-only fashion, the texture cache memory allows those chunks to be accessed more rapidly and reduces the latencies induced by uncoalesced accesses. Such texture cache is highly optimized for 2D and 3D memory reading and can be very useful for algorithms where data is read from neighbouring spatial locations.

- The constant memory is able to store a few dozen Kilobytes of data and is able to rapidly transfer its content to SM. Transferring data to constant memory is however slow. This memory is then useful for global parameters shared between threads.

- The shared memory is a block of dozens of kilobytes (typically 64kB) embedded in each SM. This memory is accessible by all the threads running on a given SM. As shared memory is highly optimized for low latency data transfer between threads, this memory is practical for reduction algorithms (summing, max, sorting) and local atomic operations. Data in shared memory is organized in 32-bit memory banks. There are 32 banks

per SM, allowing each bank to store multiple 32-bits values. Accessing different values in the same bank is inefficient since it leads to bank conflicts. Shared memory is nevertheless efficient for coalesced accesses or for broadcasting values over multiple threads.

- There are also thousands of registers per SM on which computations are directly applied. For recent architectures the number of registers can reach 65536 per SM, and 255 registers can be assigned to each thread. Registers are 32-bits in size.

On the software side, functions are implemented from a thread perspective and are named kernels. A kernel describes all the computations done by a single thread. Kernels are launched on the GPU following a launch-configuration given by the developer. The launch-configuration informs the GPU on how the computations are organized across threads, blocks and grid. Blocks are a group of at most 1024 threads which are run on a SM and can hence access the same shared memory chunk. Threads can be organized over three dimensions x, y, and z, with the product of the three dimensions being the total number of threads in that block. The grid informs the GPU on the number of blocks that must be launched, and is also organized in three dimension. The division into three dimensions is useful for multi-dimensional algorithms. This organization is important since data transfers between threads and synchronizations can only occur within the same block. Indeed, it is impossible to have a global synchronization barrier between blocks., i.e. for a single kernel launch, threads belonging to one block are unable to communicate any data with the threads from other blocks. If such global synchronization is required, the kernel have to be launched multiple times, inducing launch overheads.

From these considerations, a few golden rules can be raised. First, data transfers between the different GPU memory layers have to be limited. Access patterns should be performed in an aligned and coalesced fashion. Also, high throughput computations must be privileged. This is often reduced in a single rule of parallel computing: hide the latencies.

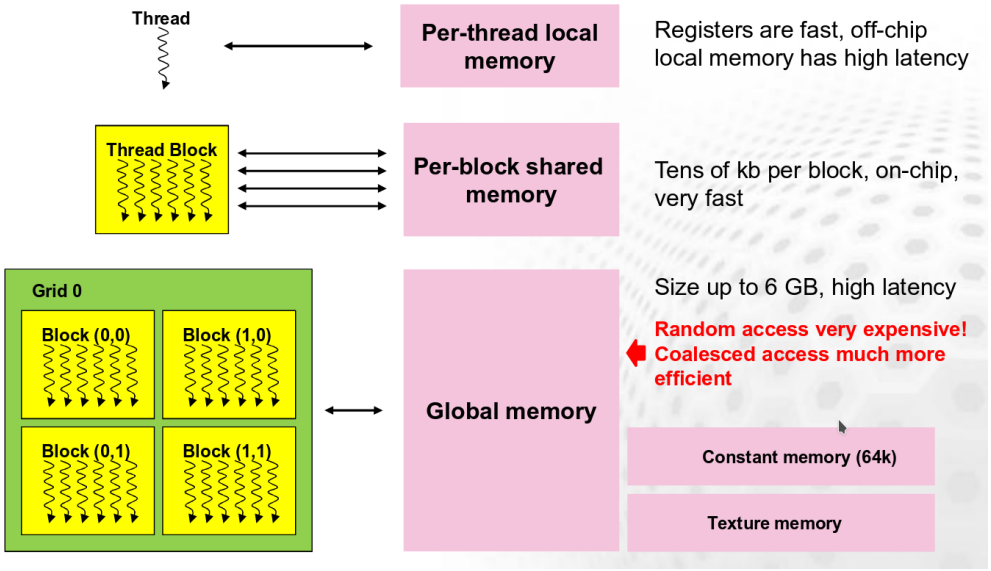


FIGURE 1.5: The CUDA threads organization across the three levels (single thread, block and grid) with their respective memory caches. ((Image from http://www.training.prace-ri.eu/uploads/tx_pracetmo/introGPUProg.pdf under the Creative Commons 3 license)

1.4 State-Of-The-Art

Recent advances in Deep Learning

Generative adversarial networks Generative Adversarial Networks (GANs) (Goodfellow et al., 2014) have been said the most interesting idea in machine learning in the last decade according to Yann LeCun. GANs are a method for learning a generative model with an adversarial process, a mini-max game between two networks: a generator and a discriminator. The discriminator is trained for a classification task between real and fake data distribution. The generator's purpose is to generate samples which belong to the real data distribution by using the discriminator gradients. The metaphor of two networks trying to fool (or beat) each other illustrate the adversarial nature of such method. This method has become popular for its ability to produce very realistic samples from different image datasets.

Since GANs were first proposed in 2014, research on GANs have become numerous (hindupuravinash, 2018). Many researchers focus on stability problems of GAN. In original research, the generator or the discriminator can become better at its specific task than the other, leading to a point where the whole network does not learn anything else. Also, GANs suffer from the mode-collapse issue, where generated images can lack in variation or worse, the generator may

learn some specific samples from the training dataset. This was adressed first with DC-GAN (Radford et al., 2015), which proposed a very stable architecture for image generation, as well as showing that features learnt by GANs are relevant for unsupervised pre-training. It was able to generate faces and house interiors realistically. Improvements in GAN training (Salimans et al., 2016) have leveraged stability and generated image sizes from 64 pixels to 256 by proposing feature matching, minibatch normalization and avoiding sparse gradients in their models. WGAN (Arjovsky et al., 2017), Improved-WGAN (Gulrajani et al., 2017) and DRAGAN (Kodali et al., 2018) propose other variation of the loss function based on the Wasserstein Distance (Vaserstein, 1969). Experimental results have shown improved stability and reduced mode-collapse.

By mixing GANs and auto-encoders architectures, applications involving artificial data generation have been widely explored. For instance:

- Faces generation (VAE-GAN (Larsen et al., 2015), DC-GAN (Radford et al., 2015))
- Super Resolution (SR-GAN (Ledig et al., 2016), 512 pixels faces (Karras et al., 2017))
- Image to Image translation (CycleGan (Zhu et al., 2017), StarGan (Choi et al., 2017))
- Realistic drawing from pixels(Pix2pix (Isola et al., 2017))
- Text to image (Reed et al., 2016)
- Music and voice generation (WaveNet (Van Den Oord et al., 2016))

Deep reinforcement learning Training AI as an agent in a complex environment is one of the most difficult tasks. Indeed, the greedy state-space exploration (with backtrack algorithm for instance) is out of question in such environments since the number of possible states needed to achieve human-like performances can rapidly explode. While greedy exploration has been shown to be efficient for some games, for instance with IBM DeeperBlue beating Garry Kasparov at chess, games like Go or current video-games face an explosion in the number of configurations and thus backtrack algorithm would take far too much time to take any decision. Instead, methods trying to approximate good solutions by learning over many iterations can overcome this issue. Such methods try to infer an action using the environment state as the input.

Evolutionary algorithms are methods based on parameters combining between the most successful agents in a population tested with a given environment. These methods are designed to mimic hereditary transmission of successful behaviours through generations. Sub-classes of evolutionary algorithms include evolution strategies (Rechenberg, 1965), evolutionary programming (Fogel et al., 1966), genetic algorithms (Fogel, 1998) and memetic algorithms (Moscato et al., 1989). Such approaches have been shown to be fairly efficient for many tasks, including autonomous car driving (Togelius et al., 2007), mobile robots (Valsalam et al., 2012) and gaming (Togelius et al., 2011). However until recently, the performance of such methods was not high enough for operational deployment.

On the other hand, back-propagation methods allow training of a single model agent through Reinforcement Learning (RL). RL is designed to mimic Pavlovian-like training. Such training uses reward and punishment signals in order to learn through experience. With the breakthrough of deep learning, Deep Reinforcement Learning (Mnih et al., 2015) (DRL) have been able to leverage current AI research in a range of tasks. Deep Q-Learning research (Mnih et al., 2013) showed near human performances on several Atari games with a single AI architecture. Deep reinforcement learning techniques also allowed the AI AlphaGo to beat the world champion of Go, Lee Sedol (Silver et al., 2016). This achievement is remarkable since the game of Go is a board game with 250^{150} possible combinations and so considered to be a very difficult game. Autonomous car-driving is also currently in the spotlight of applicative research in deep reinforcement learning (Kisačanin, 2017). While deep Q-learning can only be trained on one environment at a time, the Asynchronous Advantage Actor Critic (A3C) method (Mnih et al., 2016) recently allowed a single agent to be trained with multiple agents asynchronously, allowing scaling of the training-phase for multiple devices.

While DRL techniques seem to outperform evolutionary algorithms, recent publications show that efficient parallel implementations of evolutionary strategies can reach equivalent performance levels faster (Salimans et al., 2017). Even gradient-less approaches like genetic algorithms can be competitive with such parallelization (Such et al., 2017). We can see here that algorithm-architecture adequacy can be the main factor for optimizing performance.

One should note that autonomous agent research is currently easily accessible. OpenAI, a non-profit AI research company, developed a publicly available toolkit for reinforcement algorithms called Gym (Brockman et al., 2016). Gym

can emulate environments for Atari games, autonomous driving cars and advanced video games such as Doom and Grand Theft Auto 5. Such open-source initiatives mean that we can expect research in autonomous agents to progress rapidly.

Neural network quantization Inference with deep neural networks can be energy-consuming as computations rely on 32-bit floating-points values. The consequences are high-memory usage for data and models. In order to reduce such storage problems, a lot of research on deep neural network has looked at the impact of quantization. For example, a 32-bit value may encode up to 32 binary values, thus potentially compressing models by a factor 32. Also, in a binary framework, the dot product operation is equivalent to a bitwise XNOR and a population count operation on 32 values at the same time, resulting theoretically in sixteen times fewer computations. Finally, as shown by Seide et al. (2014), gradients quantization down to 1-bits can also be advantageous for reducing network transfers between different devices during learning, allowing better scaling during this phase.

One way to perform parameter quantization is by reformulating deep learning in a probabilist framework. By doing so, Expectation backpropagation (Soudry et al., 2014; Cheng et al., 2015) can be applied instead of the usual gradient-descent error backpropagation, allowing the training of multi-layer perceptron networks with binary weights. Networks trained with expectation backpropagation show performances equivalent to full-precision networks on the MNIST dataset. However, Expectation Backpropagation has not been applied on deep convolutional architectures. Whether it is possible or not is still an open question.

Many recent publications show that training quantized neural networks with backpropagation is possible. The BinaryConnect method (Courbariaux et al., 2015) proposes to train neural networks with binary networks by using the binary projection of real-value weights during the feedforward and backward steps. The binary projection is performed by applying the hard-sigmoid function, which is linear between -1 and 1, equal to 0 for input values lower than -1 and equal to 1 elsewhere. Once the gradients with respect to the binary weights have been computed, the real-value weights are updated with respect to the partial derivative of the hard-sigmoid function. With such methods, BinaryConnect networks show equivalent performance levels to networks with real-valued weights.

BinaryNet (Courbariaux et al., 2016) is an extension of BinaryConnect in which the same principle of binary projection is applied to activations. A neural network trained this way has all its activations and weights binarized. The architectures rely on successive layers of convolution, batch normalization and hard-sigmoid (or hard-tanh in some cases). However, quantization of both activations and weights induce a substantial performance loss on classification tasks such as MNIST, SVHN and CIFAR-10. For operational deployment, only the binary weights and batch normalization learnt mean and variance values need to be kept, hence resulting in effective compression of the model. Also, this article reported an acceleration factor of 7 with an unoptimized GPU kernel. Since batch normalization parameters are constant at inference, this step as well as binarization can be theoretically performed within the same kernel as the convolution, hence resulting in further acceleration factor due to the reduced memory overhead. Also, this architecture has been implented on different architectures (CPU, GPU, FPGA and ASIC) in order to evaluate the potential acceleration factor on these different architectures (Nurvitadhi et al., 2016). Not surprisingly, ASICs deliver four orders of magnitude acceleration factor, FPGAs three orders of magnitude and GPUs two orders of magnitude. The paper discusses whether better fabrication processes for FPGA as well as hardened parts and Digital Signal Processor (DSP) may close the gap with ASIC in terms of acceleration and energy-efficiency.

In order to extend quantization to different resolutions, the DoReFaNet framework (Zhou et al., 2016) proposed different projection functions for weights, activations and gradients which allows any resolution, which is defined prior to training as a hyper-parameter. The article shows that performance levels equivalent to full precision networks can be achieved with the AlexNet architecture (Krizhevsky et al., 2012) using 1-bit weights, 2-bit activations and 6-bit gradients on the ImageNet classification taks. It also shows that different tasks require different resolutions, since further quantization was achieved on the SVHN dataset.

The XNOR-Net method (Rastegari et al., 2016) allows deep neural networks to be trained using binary activations, weights and gradients. It approximates float-based convolutions with a binary convolution and an element-wise product busing a scalar α (the average of absolute weight values) and a 2D matrix K (the spatial mean of the input over the kernel fan-in). Gradients are computed during the backward step with a hard-tanh projection. With an additional floating-point overhead compared to BinaryNet, performances are almost equivalent to

full-resolution networks. The authors also report an effective 32 fold memory saving and 58 times faster convolutional operations.

Ternary networks training (Zhu et al., 2016; Deng et al., 2017) has also been studied. Compared to binary networks, ternary networks reach the same level of performances as their floating-point equivalent networks. Ternary networks are less difficult to train and perform better than binary networks in general.

The state-of-the-art in neural networks quantization shows that binary networks can drastically accelerate processing times although this is at the expense of performance. Keeping some information as floating-point values or having a ternary quantization instead can reduce the performance loss. In any case, the quantization parameters have to be chosen considering a trade-off between acceleration and performance. One should note that in all the previous research, only classical feedforward architectures have been benchmarked. Advanced architectures such as Residual networks (He et al., 2016), Inception networks (Szegedy et al., 2017), recurrent architectures (GRU and LSTM) and GANs have not been trained with such quantization for now. Since these advanced architectures reach the state-of-the-art, current quantization-schemes also suffer from architectural limitations impeding the best models to be accelerated this way.

Advances on large-scale simulation technologies

Available software for Deep Learning The first efficient and modular implementation of convolutional neural networks with GPU support, cuda-convnet, came from Alex Khrizhevsky prior to his victory in the ImageNet competition (Krizhevsky et al., 2012). The software architecture adopted relied on a list of layers defined in a configuration file, which processes batches of images and updates their parameters iteratively. This architectural design only allowed a single input and output per layer, but the development of the Caffe framework (Jia et al., 2014) allowed the definition of a global graph of layers. Caffe became very popular thanks to the abstract classes it proposed, allowing much more modularity and custom implementations. Caffe is written in C/C++ with CUDA support and different high-level wrappers (Python, MatLab), hence demanding advanced development skills if used in the context of research at the level of layers. Nevertheless, the attained trade-off between modularity and computational efficiency of Caffe made the framework popular in both academic and industrial communities.

Before this neural networks revival in 2012, libraries such as Theano[1] (Bergstra et al., 2010) and Torch (Collobert et al., 2002) had been developed for machine learning purpose. Theano is presented as a mathematical expression compiler for Python language. It can perform efficient vector-based computations in the same way as the Numpy library and Matlab, and perform code compilation and optimization at run-time in order to maximize the speed of processing. Compilation is performed when creating functions, which take as parameters a set of input and output place-holders as well as an optional parameters update expression (typically used for updating weights in neural networks at each step). Development with Theano is oriented toward functional-programming, where a function is a graph of transformations given one or several input tensors. Theano targets researchers more than Caffe since the first one allows tensor programming while the second allows layers programming. One should note that the Torch library also allows such tensor-based programming but does not perform run-time compilation yet.

The functional design of Theano and Torch with convenient layer-based classes inspired the many industrially-developed frameworks for machine learning. Among these libraries we can mention TensorFlow (Abadi et al., 2016) , Pytorch (Ketkar, 2017), Nervana, MXNet (Chen et al., 2015), CNTK (Seide and Agarwal, 2016) and Caffe2 (Goyal et al., 2017). As of today, Tensorflow is the framework with the most active community. Keras (Chollet et al., 2015) is also worth a mention since it provides a high-level abstraction for training deep neural networks using Theano and Tensorflow back-ends, allowing both code portability between the two frameworks and fast-development. We can also note that a major part of these frameworks are backed by industrial groups and released with a free and open-source license on Github, allowing for fast improvements with active participation from the community.

NVidia also provides active support for their GPU-devices. As AlexNet popularized NVidia GPUs with cuda-convnet (Krizhevsky et al., 2012), the company released in 2014 the cuDNN library, which provides optimized functions for standard deep learning. CuDNN is extensively integrated in almost every deep learning framework. NVidia also developed TensorRT, a C++ library which quantizes floating-point networks trained with Caffe to 8-bit integer versions that accelerate inference by a factor of up to 48.

Since deep learning literature has grown rapidly, proposing several methods,

[1]The Theano framework's develoment has been discontinued in 2017

software development for machine learning is essentially focused on the implementation of these novel methods, as well as convenient abstractions and distributed computing to facilitate learning and inference.

Dedicated hardware for Deep Learning As NVidia GPUs are the main hardware for deep neural networks , the company proposed in the latest Volta architecture (Tesla V100) dedicated computation units for $4 \times 4 \times 4$ matrix multiply-add operations using 16 to 32-bit floating-point values. This results in 64 floating-point multiply-add per clock cycle per core, with eight cores per SM. NVidia claims this dedicated units achieve an acceleration of 8 per SM compared to the earlier Pascal GP100 architecture. Considering V100 GPU has more SM and more cores per SM, the achieved increase in throughput is twelve times compared to the previous generation.

Google has also developed its Tensor Processing Unit (TPU) technology (Jouppi et al., 2017) to accelerate deep learning inference for its cloud offers. Like GPUs, TPUs are PCI-E boards with an ASIC specialized for TensorFlow routines, particularly for neural network processing. The board includes a DDR3 RAM memory for weight storages as well as local buffers for storing activations and accumulations. The dedicated units are specialized for matrix-multiplication, activation functions, normalizations and pooling. Google reports the device can perform 21.4 TFLOP/s (averaged over diverse neural architectures) for 40W power-consumption.

NVidia has also proposed solutions for embedded GPU architectures, the first generation being the Tegra K1 (TK1) with 256 CUDA cores connected to an ARMv15 processor. The second generation, the Tegra X1 (TX1), features 512 cores embedded on the device delivering up to 313 GFLOP/s for a maximum power consumption of 15W. It is possible to combine TensorRT models on TX1 to run computer vision applications in real-time in an embedded system. An alternative way to embed deep learning based applications are provided by Intel (with its Movidius chips) and GyrFalcon Technology. The two companies propose specialized USB devices which contains dedicated compute units. For instance, the Intel Movidius (Myriad 2) device embed 2Gb of RAM memory with twelve vector-processors (sixteen for their latest product, the Myriad X), which are units specialized in 128-bit based computations (able to process four floats or integers, eight half-float or sixteen 8-bit words in parallel). USB devices consumes 1W of memory for 100 GFLOP/s and are fully compatible with nanocomputers like Raspberry Pi and Odroid. In terms of performance per

watt and prices, the USB-device solution for embedded deep learning applications seems the best as of today.

Available solutions for SNN simulation Several frameworks are available for simulating large-scale spiking neural networks. Software frameworks examples: NEURON (Carnevale and Hines, 2006), GENESIS (Wilson et al., 1989), NEST (Hoang et al., 2013) and BRIAN (Goodman and Brette, 2009). These frameworks are designed and maintained with modularity and scalability purposes. They are able to handle simple neural models such as LIF and Izhikevich Neurons, complex ion-channels dynamics and synaptic plasticity. The database ModelDB (Hines et al., 2004) proposes a collection of models shared between researchers, using such frameworks.

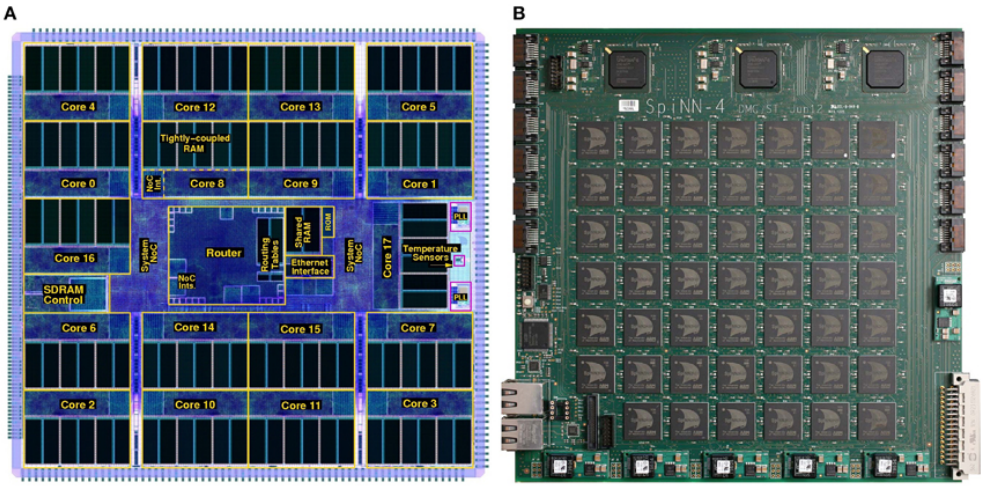
Because spiking neural networks simulations are directed toward biological plausibility, models tend to be far more complex than those in Deep Learning. Hence processing times are critical for research efficiency, mainly because of the difficulty to tune SNN dynamics parameters. For such work, methods based on genetic algorithm or Runge-Kutta optimizations are used, but require several simulation run before convergence.

Many efforts were deployed in order to accelerate processing for such simulations. NeMo (Fidjeland and Shanahan, 2010), BRIAN, Nengo (Bekolay et al., 2014), GeNN (Yavuz et al., 2014) and CarlSim (Beyeler et al., 2015) have GPU-processing capabilities. Brian, Nengo and NEST also have distributed computing options, allowing large simulations to be run on clusters. Given the different features of the all these simulation frameworks, PyNN library (Davison et al., 2009) allows a unified approach of SNN model definition and makes it possible to run models on different frameworks (currently wih Brian, Nengo and NEST), but also on neuromorphic devices and specialized clusters.

Neuromorphic hardware While the brain process information through billions of computation units in parallel, nowadays computers rely on the Von Neumann architectures, where computations are performed sequentially with an externalized memory accessed from a bus. The difficulty of running large-scale simulations of biological neural networks comes from this drastic difference between the brain and the Von Neumann architectures. In this sense, a lot of research is directed toward neuromorphic architectures in order to reach the speed and energy efficiency observed in the brain.

One system, developed in 2014 in the context of the European Human Brain Project (Markram, 2012) is SpiNNAker. The Spinnaker hardware (Furber et al., 2014) is based on digital ARM-cores organized in clusters in order to simulate very large-scale neural networks. A SpiNNAker system can be scaled from 1 to 600 chips, each being able to simulate 16000 neurons and eight million synapses in real time using a single watt per chip. A single SpiNNAker chip contains 18 ARM cores at 200 MHz clock frequency with 32KiB for instructions and 64 KiB for data TCM memory modules. A core connects all its embedded neurons in an all-to-all fashion. Each chip also has 128MiB of SDRAM for synaptic storage, as well as a multicast Network-On-Chip which allows spikes to be transmitted to all cores and chips connected using Adress Event Representation (AER) encoding (each neuron is addressed using a unique 32-bit identifier). Each chip can be connected in a 2D fashion to six other chips, hence chips are organized in a grid where each chip communicates with its six neighbours. For connections between non-neighbouring chips, spikes have to cross multiple multicast units in order to reach the target neurons. SpiNNaker was designed with two principles in mind, scalability through the multi-chip grid architecture, and energy-efficiency with the use of ARM cores, resulting in a single watt consumption per chip. SpiNNaker applications remain scattered across literature. Some successful studies have managed to provide test applications or simulations running on SpiNNaker devices. Sugiarto et al. (2016) show an example of image processing on SpiNNaker, having a Sobel edge detector module running at 697 FPS on 1600×1200 images. Mikaitis et al. (2018) proposes a pavliovian conditionning neural simulation inspired from Izhikevich's research (Izhikevich, 2007). In this research, the neural plasticity model could be run in real time. While the real time constraint is achieved in all conditions (regarding the number of synapses), the SpiNNaker system with less than 7 million synapses performs not better than an NVidia TX1, an embeddable GPU chip with 512 CUDA cores. The SpiNNaker chip is the first large-scale attempt at running huge neural simulations *in silico* with particular attention toward scalability, but more applicative research is required in order to figure out in which domains this chip can be relevant.

Another hardware project developed as part of the Human Brain Project hardware is BrainScaleS (Kunkel et al., 2012). The BrainScaleS hardware simulates AdEx neurons (Brette and Gerstner, 2005) and features dynamic 4-bit synapses. Based on wafer-blocks containing High Input Count Analog Neural Network chips (HICANNs). Each wafer can simulate 44 millions synapses and 192608 neurons and can be connected to up to 20 other wafers. BrainScaleS



The SpiNNaker chip A 48-node SpiNNaker board (SpiNN-4)

FIGURE 1.6: On the left, the architecture inside a single SpiNNaker core. On the right, the mesh organization of several SpiNNaker on board ((Image from Lagorce et al. (2015) under the Creative Commons 4 license)

allows processing times from 10e3 to 10e5 faster than real-time. However, the literature lacks any applications related to machine learning using this device, hence no conclusion about its use-cases can be drawn.

IBM's TrueNorth chip (Merolla et al., 2014) is a CMOS integrated circuit containing 4096 cores. Each core can process 256 neurons linked in a fully-connected manner with a crossbar array, for a total of million simulated neurons and 256 million synapses on the chip. Every neuron can be configured, and receives its input from an axonal pool common to every neuron in the circuit. When a spike is emitted, it is added to the axonal pool of spikes, which are sorted at the core-level. All the cores are connected in a 2D array of 64-by-64 cores, and spikes are propagated first through the x-axis of the mesh, then through the y-axis. A local-router and an axonal look-up table is embedded within each core in order to send spikes efficiently and asynchrounously. The TrueNorth hardware has received more attention from an applicative point-of-view compared to the previously cited hardware (866 citations for TrueNorth, 245 for SpiNNaker and 106 for BrainScaleS). Example applications concern mainly deep learning acceleration Esser et al. (2016); Diehl et al. (2016b,a). For instance, Esser et al. (2016) managed to adapt classical CNN architectures for a cluster of TrueNorth chips in order to reach near state-of-the-art performances for an energy consumption reduced by a factor 100. However, using a single chip degrades performances, and the quantization scheme still induces a

reduction of performances on the cluster. Also, the input dimensions are relatively low, with $32 \times 32 \times 3$ pixels for images and up to $32 \times 16 \times 3$ for audio, compared to Deep Learning tasks with dimensions ($256 \times 256 \times 3 \times 32$). For the TrueNorth architecture, energy efficiency seems to be the main design criterion, sacrificing some performances and memory resources. Since this project is assumed by IBM to be a first attempt at building efficient neuromorphic devices (IBM, 2015), improvements on such architectures can reasonably be expected from the firm. For instance, TrueNorth's fabrication process is 28nm while GPUs arenow using 14nm technology (7nm achievable in 2019) and synaptic plasticity on-chip has been announced.

Finally, Intel announced recently (end of 2017) the release of the LoiHi chip, a wafer-based hardware device able to simulate 130 thousand neurons and 130 million synapses. The LoiHi architecture uses a 128 neuromorphic cores mesh, each core being able to simulate 1024 neurons with an embedded self-learning module. The LoiHi chip supports essentially any communication scheme, i.e. unicast, multicast, and broadcast, as well as sparse network, any synaptic resolution from 1 to 9 bits and hierarchical connectivity. The latest feature promises to be efficient to avoid the need for global synchronization of spikes packets. The LoiHi chip will also be fabricated with a 14nm process. As the LoiHi chip has not been released yet, any announced performance remains speculative.

Bridging the gap between Machine Learning and Neuroscience

As shown in (Esser et al., 2016), converting CNNs trained with backpropagation to SNNs can be done to take advantage of the hardware energy-efficiency of the latter technology. Recurrent neural networks are also convertible in such way (Diehl et al., 2016b) (Diehl et al., 2016a). These studies show that, for inference at least, SNNs can extend the formalism of CNNs and process information equally.

Yet, reaching current CNNs performances on machine learning tasks with pure SNN learning mechanisms remains a challenge as of today. Research in biological networks is mainly focused on the question "How does the brain learn?", while machine learning researches targets at understanding intelligence independently of the physical medium. Some research however tries to understand learning by the combination of both literatures.

The STDP mechanism often used in SNN acts as a coincidence detector between input events (Guyonneau et al., 2005; Bender et al., 2006; Karmarkar

and Buonomano, 2002). Further research show that in the context of vision, STDP allows feedforward spiking neural networks to learn early visual features and representations (Masquelier and Thorpe, 2007). Kheradpisheh et al. (2016) also show that a two-layers SNN with STDP and pooling-like Winner-Takes-All mechanism can learn visual features relevant for image classification. By applying STDP to feedforward connections and anti-STDP on feedback connections, spiking neural network are able to minimize an autoencoder loss function (Burbank, 2015). By using STDP, spiking neural networks are thus able to perform unsupervised learning. How SNNs perform supervised learning is however an active question. Indeed, backpropagation seems non-biologically plausible since there is no strong biological evidence of such global error-signalling in the brain.

Izhikevich proposed a spiking network model in which the credit assignment problem is tackled through interactions between STDP eligibility traced in synapses and error-dependent dopamine releases (Izhikevich, 2007). While this type of model has not been tested on complex machine learning tasks, it is able to generate spike patterns learnt in a supervised manner (Farries and Fairhall, 2007). Further experiments with a four-layer SNN with conditioning signals have achieved state-of-the-art performances on simple yet non-trivial visual classification tasks (Mozafari et al., 2017). Such results were obtained applying STDP learning rule in the first three layers while the last benefits from reward and punishment information. As the error-signal is not backpropagated through the whole architecture, neuromodulator-based supervised learning does not constitute evidence for the existence of a separate error pathway.

Nevertheless, STDP has captured the interest of the machine learning community. Bengio et al. (2015) showed that STDP occurring in a single layer is equivalent to gradient descent method and to a biologically plausible implementation of Expectation-Maximization algorithm. The main requirement for biological networks to support backpropagation is the existence of feedback connections between neurons in the hierarchy which could support error backpropagation. Lillicrap et al. (2016) shows that if the existence of a feedback connection for each feedforward one is mandatory to support such hypothesis, the weight-symmetry condition of these two connections is not. Hence a neural network with reciprocal but not symmetric feedforward-feedback connections is able to support backpropagation. Such reciprocal connections could potentially be supported by a segregated dendritic pathway (Guerguiev et al., 2017), with basal dendrites for feedforward connections and apical dendrites for feedback connections (Spratling, 2002; Budd, 1998).

Finally, recent interactions between microglia and neurons through neurotransmitters (Glebov et al., 2015) show that the microglia intervene in the regulation of synaptic changes in function of the activity. The role of microglia has long been attributed to immune functions only, thus not affecting the activity of neurons themselves. However, such novel interactions in activity regulation may revive the debates on the non-plausibility of backpropagation since it might be consistent with the separate error pathway hypothesis.

1.5 Aim and contributions

Deep Learning methods have been able to become successful at many tasks in machine learning thanks to two main factors: the addition of constraints which find equivalent mechanisms in biology and an adequacy algorithm architectures for GPU hardware. All the proposed priors can easily be implemented on such platforms. Moreover deep neural networks use simple computations compared to those used to simulate biological neural networks. We argue that current parallel implementation are focused on fairly complex models of spiking neural networks, hence such implementation does not allow competitive processing times for industrial applications. We emit the hypothesis that parallel implementation of slightly simpler models of spiking neural networks can be shown to be efficient for such purposes. We have also seen that the neuroscience literature contains many mechanisms in biological neural networks that remain barely explored by the machine learning community. We emit a second hypothesis that these mechanisms can accelerate further inference and learning in application-oriented neural networks, CNNs or spiking neural networks (SNNs).

We propose three contribution in this thesis to address these hypotheses.

- The first contribution consists in the study of the algorithm-architecture adequacy of BCVision, an industrial grade visual pattern detection model based on spiking neural networks and rank order coding, for GPU architectures. We also study the effect of the addition of a hierarchical coarse-to-fine processing scheme on processing times. Our main question in this contribution is: how much can we accelerate rank-order coding neural networks with parallel hardware implementations and coarse-to-fine processing.
- The second contribution aims at comparing different spike propagation algorithms on GPU architectures. We show that considering a neural layer

hyper-parameters and hardware specifications, we are able to predict processing times for three algorithms, then compare the algorithms in order to spot cases where each of them is the most efficient. The computational models deduced from this work can be used to predict the efficiency of a dedicated hardware device for a given neural network architecture.

- In the third contribution we adapted the STDP learning rule for single feedforward based learning of visual features. We propose a model of early visual feature learning based on a binary version of STDP with stabilization mechanisms using only one spike per neuron per sample. We show that qualitatively the features learnt match the preferred receptive field of V1 layer neurons. We also show that such features, which are learnt from ten to hundred times less data than classical Deep Learning approach, reach state-of-the-art performances in four classification tasks.

Chapter 2

Algorithm / Architecture Adequacy for fast visual pattern recognition with rank order coding network

2.1 Algorithm / Architecture Adequacy of BCVision, an industrial ultra-rapid categorization model

2.1.1 BCVision : an industrialized model of Rapid Visual Categorization

The speed of processing in rapid visual categorization tasks is very high as shown by Thorpe et al. (1996). The main conclusion of this article is that the neural latencies observed in visual cortex of primates during such tasks allow a single feedforward neural propagation phase in order to perform fast but accurate visual categorization. Such observed reaction times also allow only one spike per neuron to be fired during a single categorization. These conclusions led to research that demonstrated successful face recognition using only one spike per neuron (Van Rullen et al., 1998; Delorme and Thorpe, 2001). The SpikeNet simulator was designed with the hypothesis of a single feedforward propagation exploiting the intensity-latency equivalence that only needs one spike per neuron to be fired for each image. This simulator was improved for

development of industrial application (Thorpe et al., 2004), with the improvement of BCVision, a software library developed in 1999 by SpikeNet Technology [1] based on SpikeNet simulator (Delorme and Thorpe, 2001).

BCVision can perform one-shot learning of visual patterns. To learn a new model, the user must select an image patch containing the reference pattern to the application. Learnt patterns are also called models in the following section. Once one or more patterns have been learned, BCVision can perform online visual pattern detection in order to retrieve the position of the learnt patterns. The different BCVision use cases are presented in Fig. 2.1.

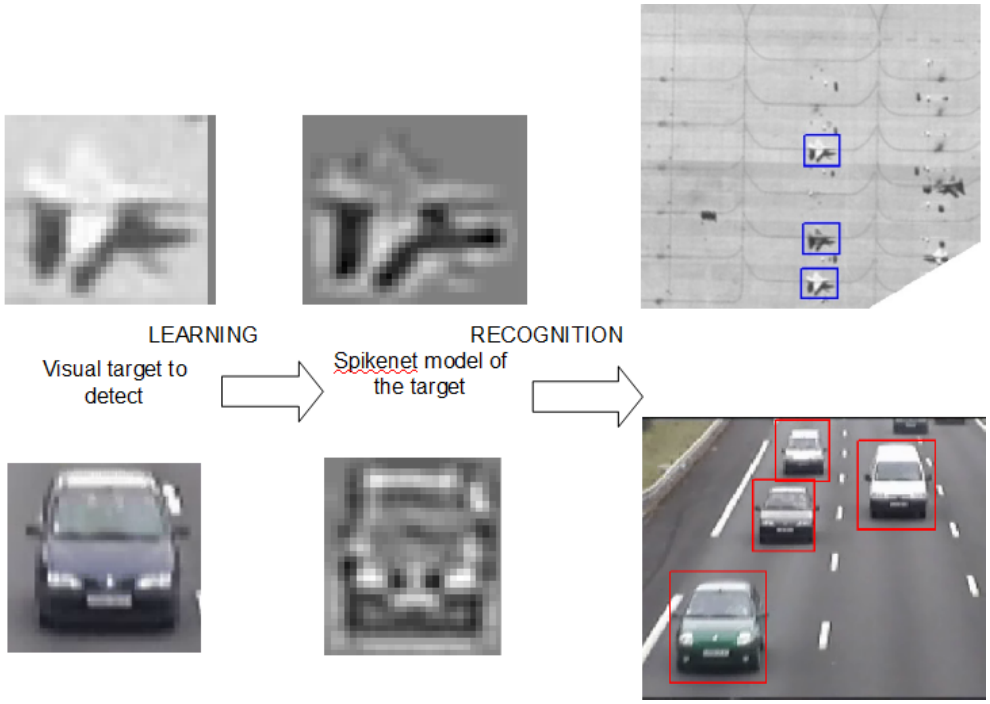


FIGURE 2.1: Principle of BCVision use cases

Learning a new model is first performed by rescaling the reference image in such way the rescaled image has an area of 900 pixels, i.e. for a input image with an aspect ratio of 1 this will result in a 30x30 rescaled image. Spike generation is then done by filtering the rescaled image with oriented gabor like filters and sorted by rank of magnitude. Finally, a few percent (which can be adjusted depending on the use case) of the first spikes are kept to define the model.

The recognition phase, also called pattern detection process can be divided in four main steps:

- First, the input image is resized multiple times in order to detect the learnt models at different scales.

[1] SpikeNet Technology is now BrainChip SAS, a BrainChip Holding LTD Company.

- Secondly, each scale (with size $W \times H$) is used to generate spikes as in the learning phase and the resulting map is thresholded in order to obtain binary spike-maps of size $W \times H \times 8$.
- Thirdly, for each model, the corresponding neuron model layers are stimulated to generate a potential map.
- Finally, each potential map is then thresholded. Every position containing a one on the thresholded potential map is considered as a detection of its respective model. A bounding box is returned for each detection, computed from the coordinated in the map and the scale on which the model was detected. Note that the choice of the threshold directly affect the detection performances of BCVision, since a lower threshold will promote recall over precision will a high threshold will return accurate results but with a low recall.

The third step, called spike propagagtion, is done mainly comparing the model selected spikes with the current scaled image spikes. To do so, for each spike in a spike-map, the potential map is incremented according to the translation coordinates in the model for the given spike orientation. Once all the spikes have been processed, we obtain a potential map of size $W \times H$ for each model.

The different processing steps performed by BCVision during recognition are presented in Fig. 2.2.

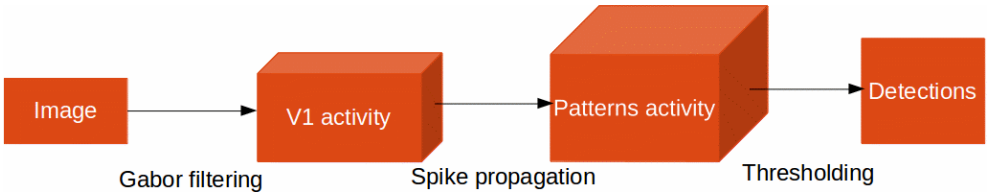


FIGURE 2.2: Architecture of BCVision kernel process for a single image scale.

Because the models are 900 pixels in size (30x30), the application takes advantages of the multiple scales to quickly detect objects. However, searching for small objects in large images (e.g. 1920x1080 full HD images) is costly in processing time. For instance on a 2GHz x86 core, detecting a single 30x30 pixel object in a HD image takes 120ms in order to perform the BCVision recognition step. Furthermore, seeking to identify a large number of models increases the processing time, since the propagation step is repeated for each model. As GPU architectures are specialized in parallel processing on large matrices, we wanted to study the suitability of an implementation dedicated to GPU-type

processors in order to speed up processing and push the technical limits of the application. Our goal at the end of this study was to process as many models as possible on a 1080p HD image in real time.

2.2 An efficient GPU implementation of fast recognition process

2.2.1 BCVision enhancement with GPU architectures

Adapting BCVision individual steps for GPU processing

The first part of this study was to study for each step of the recognition process, the adequacy of GPU and then get an efficient implementation with CUDA.

The first step being a simple scaling algorithm (bilinear filtering), where each output pixel is processed independently, GPU are well-known to efficiently parallelize the processing per output pixel. The second step (generation of spikes maps), relying on filtering, also processes output pixels independently. Moreover, since the eight filters are fixed, they can be stored in the constant memory of the device in order to optimize further the memory accesses. The CUDA implementation of these two steps was thus straightforward.

The difficulty of optimization was the parallelization of the third step (spike propagation through neuron model). In the CPU version, this step is done iteratively. Thus, given the constraints explained in 1.3, such iterative algorithms has to be modified to suit the GPU architecture and to get parallelized efficiently. This step is all the more critical as it is repeated for each model learned.

Finally, the last step (thresholding) consists in loading a single input value per thread, comparing it to the threshold, and storing the result of the comparison at the corresponding pixel in the output map. This type of operation is also simple to optimize.

Optimizations of the spikes propagation

The propagation stage in the original software (on CPU) proceeds as follows for each model. A potential map of size W, H is initialized to 0. For each spike, the

list of synaptic connections corresponding to its feature map index is browsed. For each synapse in this list, the corresponding weight is added to the potential map at the coordinates of the post-synaptic neuron.

In order to parallelize efficiently this propagation phase, we studied the algorithmic modifications necessary to fit with the GPU architecture. GPU programming is SPMD (Single Process Multiple Data) based. This involves studying and selecting the data for which the division into processing blocks leads to the fastest processing times. We first present a range of possible parallelization schemes. We envisaged three paralellization schemes :

- **Threading with respects to input pixels.** This parallelization schemes associate one thread on the GPU to a single input spike. Each thread has access to its affected input spike and the model matrix S. It then increments several values in the output matrix given the models. At first sight, this can be a relevant choice since considering the number of input spikes to process, parallelization on this data can benefits from the GPU. Also, accesses to input and models matrices can be done in a coalescent manner. Adding several weights to the potentials map is possible in parallel with the use of global atomic ADD operations. It is also the parallelization scheme which is the closest to the original sequential algorithm.

- **Threading with respect to the models**. Each thread is responsible here to the propagation of one pixel of a model across all its output potential map. This scheme has no advantage here since the number of model values is low compared to the number of values in the other matrices. Moreover, memory accesses remain performed in a concurrent manner with this implementation

- **Threading with respect to the output potential map values**. Each thread updates a single output potential by accessing the input and models. All the threads in a block keep a counter for the potential value they handle, and read the input spikes in their receptive field and their respective model in order to check how many spikes are matching the ones in the model. Input spikes may be read in a shifted manner respective to the model or in a sequential way reading all the input values in their receptive field. The first reading scheme can imply unaligned memory accesses but performs only relevant computations, avoiding processing zero spikes values in the model. The second reading scheme read the memory in a aligned and coalesced manner, but performs many computations on zeros

in the model. This last reading scheme is equivalent to the filtering or convolution operation.

We also applied the following optimization. Since network spikes are binary, they can therefore be stored using 1 bit on the pixel of the corresponding image. The spikes are extracted according to eight orientations so all the spikes extracted from a pixel can be gathered on a byte (char type). The weights of the models are also binary values with for each pixel bytes with 1 at the positions of the activated orientations. Potential propagation can be assimilated to the scalar product of spikes by synaptic weights: $Pot_{x,y} = \sum_{c \in C} S_c.x_c$, where C is the receptive field of the neuron, x_c belongs to X_C the vector of spikes and S is the matrix of binary synapses of the model. Since X_C and S contain binary values, the scalar product is calculated by:

$$Pot_{x,y} = PopCount(X_C \wedge S) \tag{2.1}$$

. where $\wedge$ is the bitwise AND operator, and where $PopCount$ is the bit counting operator. An illustration of this binary propagation step is shown in Fig. 2.3.

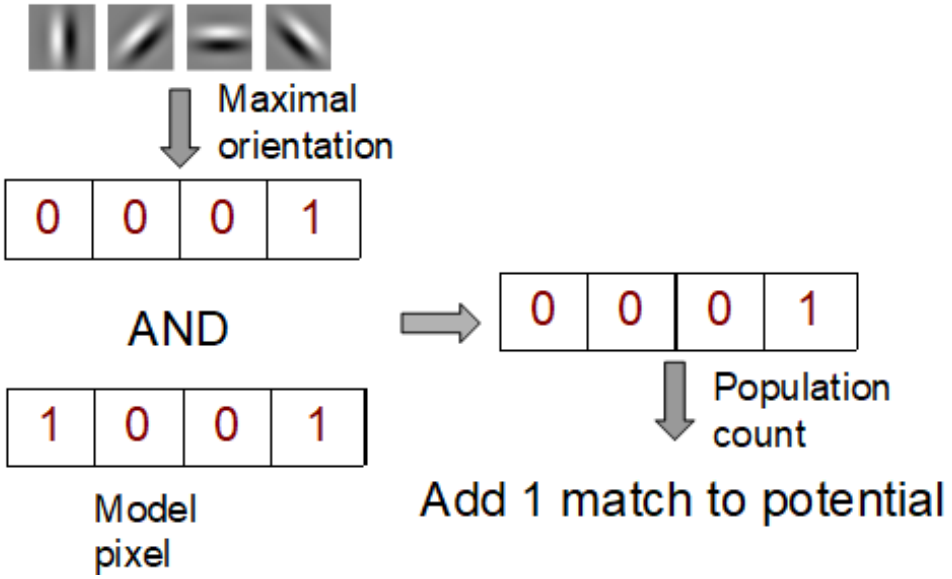


FIGURE 2.3: Binary propagation optimization in BCVision

Moreover, in order to encourage coalescent and aligned memory access, each CUDA core in the same processing block processes its neighbour's input spike to the left, logically offset by 1 byte. This way, spikes are accessed in memory as often as possible, limiting the bandwidth between processing and GPU memory. On GPU implementations, we also modify the model sizes to fit a 32-bits alignment, going from 30x30x8 models on the CPU version to 32x32x8 models.

2.2.2 Experiments and results

Comparison between threading strategies

We benchmarked the processing times of the three implementations described in the previous section on a 10 models recognition phase on a HD image.

- The first implementation is parallelized across input spikes and perform updates with global atomic addition.
- The second implementation is parallelized across output spikes and read only the input values requires by the model.
- The third implementation also parallelizes the processing across output values by performing a convolution.

We found that the first implementation accelerated the processing times compared to the original CPU implementation by a factor 2. The second implementation shown a acceleration factor of 4, while the third implementation accelerated this use-case by a factor 6.

The modest acceleration of the first method can be explained by the use of global atomic additions, which involve competing write access in a parallel implementation. In addition, the index shifts of synaptic connections imply non-coalescent memory accesses. These two implications strongly penalize the speed of processing. While caching the output potentials into shared memory can be envisaged since atomic operation on shared memory have been drastically optimized starting with the NVidia Maxwell architecture, we argue that this would results in poor performance. If the output potentials are stored in char type (8-bits words) or half type (16-bits words), accesses to different elements in the shared memory would trigger some bank conflicts, which will penalize every individual update. If the output potentials are stored in int types (32-bits integers), since a single spike can update 32x32 positions, instantiating a 256 thread block (which is one quarter of the maximum thread capacity of a block) for a 16x16 input region would require 72Kib of shared memory, which is limited to 64Kib. Caching output potentials into shared memory exhibits penalizing behaviours affecting performance with every storage type.

The acceleration difference between the second and the third method can be explained by the read access memory pattern. It seems the misalignment induced by the second method implies an important timing overhead, while in the third implementation this overhead is completely hidden by the computations

in the third implementation. Since GPU architectures rely on the principle of SPMD (Single Process Multiple Data), bottlenecks often emerge from memory latencies, which are exacerbated when the accesses are misaligned and / or uncoalescent such as in the first two methods. Hence parallelization of the computations across output potentials in a convolutional fashion seems the best choice in terms of processing times.

Through these optimizations, we have reduced the non-parallelizable GPU propagation algorithm to a simplified convolution problem, essentially based on logical operations and additions, all of which use the GPU's most powerful computing units. Multiplication, being between a value and a binary, can be performed using only logical operators.

After this first computational study, we experimented with the detection performance and acceleration obtained by massive parallelization. Three experiments were conducted to compare the initial and parallelized versions of BCVision.

Material and methods

Detection performances dataset The objective of the first experiment was to compare the detection performance of the two versions (CPU and GPU with convolutions) of BCVision in order to ensure there was no regression.

To do this, we used a generated dataset of images of patterns with distractor backgrounds. The pattern dataset used is a subset of Caltech-101 (Li et al., 2003) (classification dataset of 101 classes). the dynamic backgrounds came from the Google Backgrounds dataset (Fergus et al., 2005). Examples of the generated images are presented in Fig. 2.4.

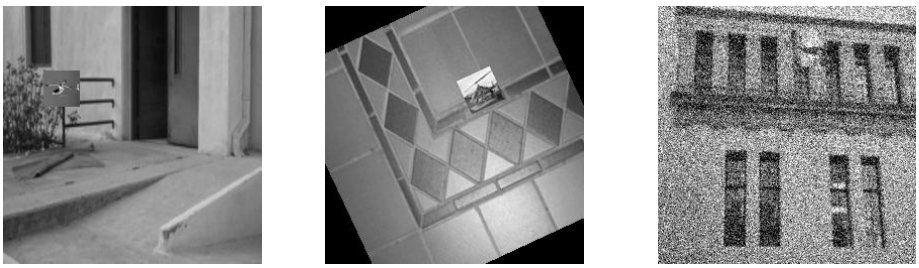


FIGURE 2.4: Example of images in the detection performance dataset. Left : control image without transformations. Center : rotated image. Right : image with white noise.

Each image of the non-regression test set has been generated pasting one pattern from Caltech101, scaled to 30x30, on one background. The generated image

may be altered with the application of an image transformation which helped us evaluate performance. Each transformation is associated to a generated subset, leading to the six following subsets. A hundred images were generated for each pattern and transformation parameter, resulting in a 600 images for our non-regression evaluation, split into six subsets.

- "Raw" subset : A control dataset without transformation of the pattern. Images are pasted with coordinated which are a multiple of the CUDA memory alignment of 128 bits, thus the top left coordinates modulus 16 must be equal to zero (given we have eights orientations). The previous point explain the distinction with the "Translation" subset described below, since the difference between these two subsets is important for the experiments in section 2.3. It gives us a performance baseline since not having a one hundred percent detection rate on this subset indicates the implementation is not even capable of matching an exact pattern.
- "Translation" subset : images are randomly pasted in the background image without any constraints on alignment.
- "Rotation" subset : images are randomly rotated in a range -30 to + 30 degrees.
- "Noise" subset : a white noise is applied on all the image. This white noise have a variance ranging from 0 to 90 percent of the considered image luminance values.
- "Contrast" subset : contrast reduction is performed in such way that the variance histogram of the pixel values is squashed between 100 percent to ten percent of its original value.
- "Scale" subset : the pattern is first rescaled by a factor ranging from 0.8 to 1.2 before being pasted in the background.

For each algorithm studied, all the patterns inserted in the images are learned. Detection is carried out several times on each image with different thresholds. The detections are then compared to the ground truth of the position of the pattern in the images. We then calculate the F-measure (more precisely the F1-score) of the detections. This score combines precision and recall rates and allow a good visualization of the overall performances of the method. For each version, we show the results obtained with the threshold which have the best F-measure. Equations for the different detction metrics are given in Eqs. 2.2,

2.3 and 2.4, where tp stands for the number of true positives, fn the number of false negative and fp the number of false positive.

$$Recall = \frac{tp}{tp + fn} \tag{2.2}$$

$$Precision = \frac{tp}{tp + fp} \tag{2.3}$$

$$F - measure = \frac{2}{\frac{1}{Recall} + \frac{1}{Precision}} \tag{2.4}$$

Time performances dataset With the second experiment, we wanted to observe the evolution of processing times through CUDA optimization. Using 1080p HD images, we have recorded processing times to detect N models, N ranging from 1 to 100. We also noted the number of models from which the processing times are greater than 33 ms, i. e. the real-time constraint for a 30 frames per second video. We used for this experiment an Intel i7-3770k for the CPU version with multi-thread processing enabled and a NVidia GTX 970 for the GPU version.

Results

We ran both versions of BCVision recognition (CPU and GPU) on both datasets. The following section describes the results in terms of detection performances for the first dataset and in terms of processing times for the second.

Detection performances The results of the experiment indicate a decrease in F-measure with the CUDA version on rotations (from 0.25 detection rate with CPU version to 0.2 for GPU), noise (0.83 for CPU, 0.72 for GPU) and scaling (0.18 for CPU, 0.12 for GPU) compared to the initial version of the algorithm. This affects the overall F-Measure with a drop of 5% in absolute performances, 7% relative to the CPU version. Decomposing the F-measure shown this drop of performances was due to a reduction of recall when the input image is heavily rotated, scaled or noised. We conclude that the modification of models size from 30x30 to 32x32 pixels affected the performances of BCVision, having the GPU version to learn more specialized models than in the CPU version. Hence on GPU in order to get the same recall as in the CPU version, the detection

threshold must be set lower, hence generating more false alarms an lowering consequently the F-measure. The results of this experiment are presented in details in Fig. 2.5.

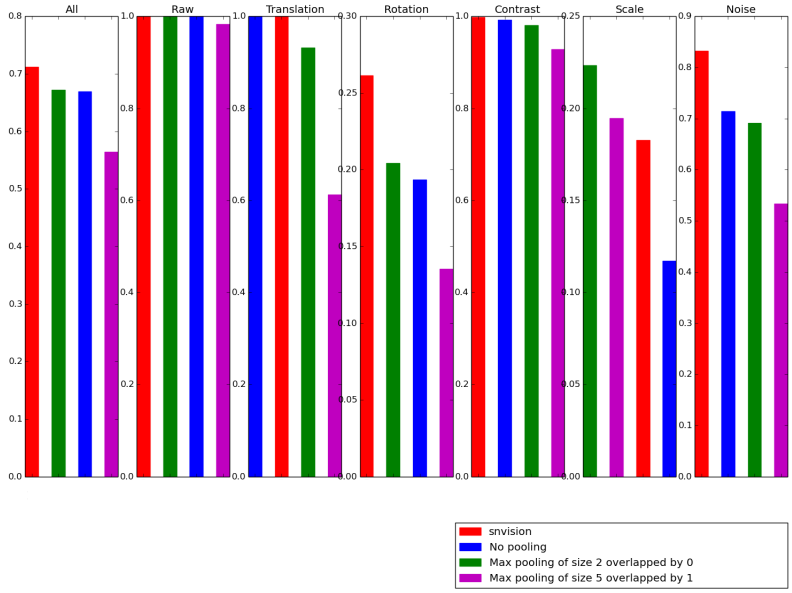


FIGURE 2.5: Detection performances of the CPU and GPU versions of BCVision on the Caltech-101 pasted images dataset. In red : the original CPU version. In blue : the GPU version without max-pooling. In green : the GPU version equipped with complex cells with stride $s = 2$. In purple : the GPU version with max-pooling with stride $s = 4$

Time performances The acceleration factor in this scenario is 6.39, allowing 15 models to be processed in real time (at 30 frames per second), while the CPU version requires 227.3 ms to process the same number of models. Note that with HD images, the CPU version is unable to reach real time performances whatever the number of models, since a single model takes 101.6 ms to be processed. Details on processing times as a function of the number of models processed along with linear regression of processing times for both conditions are available in the Fig. 2.6. Other tests have led us to delineate the advantages of GPU optimization over the standard version. For applications with low-resolution images and few models, the standard version is still more efficient, due to memory transfers and synchronizations not compensated by the number of calculations in the CUDA version. On small images (less than 128x128), the GPU version becomes faster from about a hundred models. From 512x512, the GPU version is faster from two models.

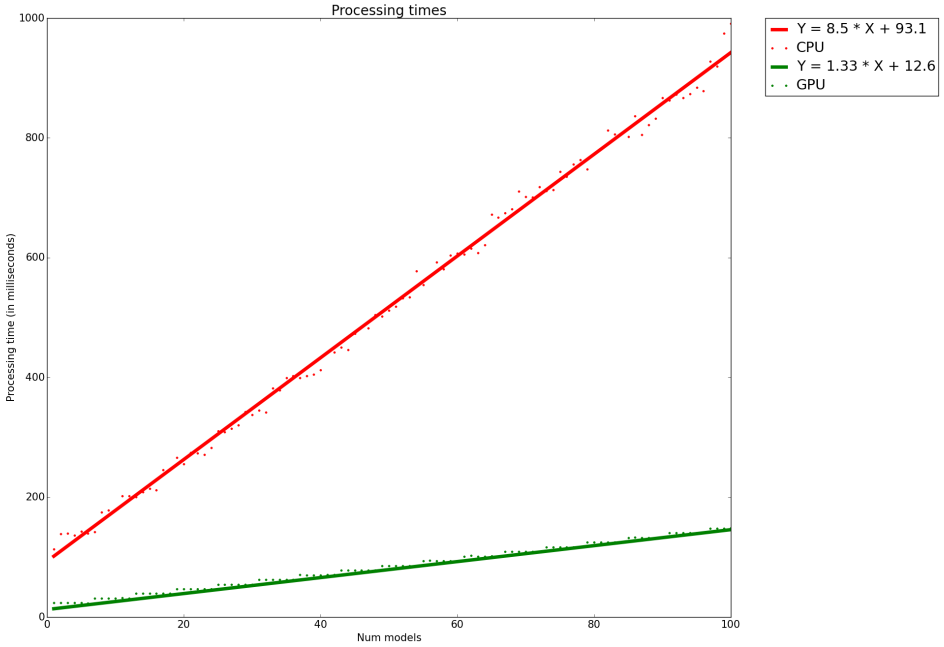


FIGURE 2.6: Acceleration factor in seconds given the number of models on a single scale 1080p image. The linear regression of processing times shows as the number of models grows the acceleration factor converges toward 6.39. When the number of models is low, this acceleration is

Conclusion

Our algorithm-architecture adequacy study of BCVision for GPU showed that the formalism of spike propagation was not adapted to a parallel implementation as such. The convolutional formalism method has proved to be more suitable for this architecture. Also the convolutional formalism allowed us to express the matching step between spikes and models as fast binary operations. All the remaining steps of the recognition phase (including rescaling, filtering and thresholding) being classical image processing operations, their implementation on GPU was straightforward. These steps have been implemented in a near-to-optimal way regarding their theoretical algorithmic complexity and profiling results

The adaptation of BCVision on GPU allowed us to benchmark this novel implementation. We tested the non-regression of the adaptation with a detection performance experiment, and recorded the processing times of both version (CPU and GPU) of the recognition phase on HD images as a function of the number of models. The proposed adaptation allowed us to accelerate processing times by a factor 6.39 although there was a drop in detection performance of 7%

due to the model size difference which lead to the learning of models requiring a lower threshold to reach the same recall as the CPU version.

While the obtained acceleration factor of 6.39 is in the order of what can be expected with a GPU implementation, further acceleration could only be achieved by more significant changes to the initial algorithm. The following section presents studies that aim to modify the algorithm to allow hierarchical detection of visual patterns, allowing greater acceleration factors while guaranteeing the non-regression of BCVision.

2.3 Coarse-to-fine approach for rapid visual categorization

In this section we propose to adapt the algorithm of BCVision to take advantage of the GPU architectures acceleration potential. We will first present different subsampling methods which allow dimensionality reduction. Next, we propose a coarse-to-fine architecture for BCVision. This novel architecture combines two networks which perform the spike propagation phase at different resolutions, in order to accelerate drastically the processing times.

2.3.1 Subsampling in the brain and neural networks

Complex cells

In the early visual cortex two main types of cells sensitive to orientation can be found (Hubel and Wiesel, 1962). Simple cells have distinct excitatory and inhibition regions within their receptive fields, and perform mathematical operations equivalent to signal filtering. They correspond to the neurons found in BCVision, and also to the perceptron neural model widely used in Deep Learning. Complex cells are fully excitatory cells which shows spatial and rotation (in a limited range) invariance characteristics due to their larger receptive fields. Complex cells are fed with spikes from simple cells, and were first modelized in the Neocognitron model (Fukushima and Miyake, 1982; Fukushima, 1989) as subsampling operators. Such complex cells can be seen as a biological means to reduce the dimensionality of inputs and to bring local invariance to translation and rotation.

Complex cells are nowadays widely used in simulation of visual processing in the brain (Riesenhuber and Poggio, 1999; Masquelier and Thorpe, 2007; Kheradpisheh et al., 2016; Mozafari et al., 2017) and in Deep Learning models (LeCun et al., 1998; Krizhevsky, 2009,?; Szegedy et al., 2015). In the latest literature, complex cells are often performed with a **pooling** operation. The basis of such operation is as follows. The pooling operation is determined by three hyperparameters : the size of its receptive field k, its stride s(the offset between two neighbouring complex neurons in the spatial dimension) and a reduction operator, basically the average or the maximum. An example of max-pooling processing is shown in Fig. 2.7. Given an input matrix X of dimension (W, H, C), where W and H are spatial dimensions and C the number of feature maps extracted by C weight-sharing simple neurons (also called convolution layer in Deep Learning), the pooling operator will apply its reduction operator on every region of size $k \times k \times 1$ offset by its stride s and output a single value for each region. Note that pooling is applied on every feature map independently. Output dimensions after pooling are $(\frac{W-\frac{k}{2}}{s}, \frac{H-\frac{k}{2}}{s}, C)$. The choice between max-pooling and average is dependent on the task and the layers' depth in which the operation takes place, as max-pooling seems more suitable to add invariance at feature levels (Scherer et al., 2010) while average pooling is often used as the last subsampling layer before the classification layer in fully convolutional architectures (He et al., 2016).

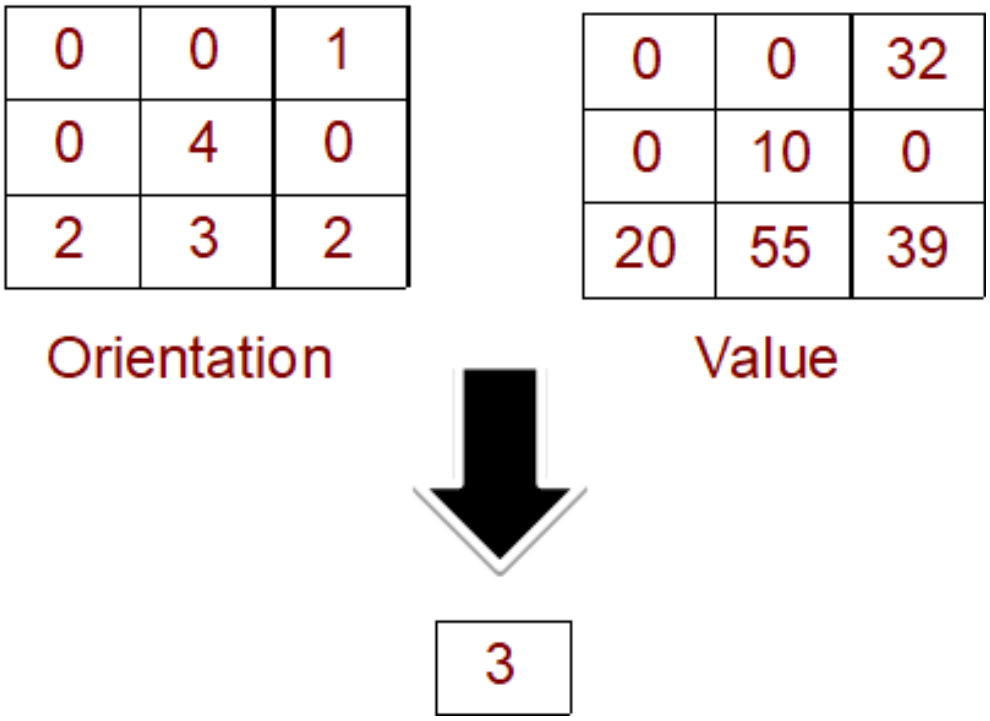


FIGURE 2.7: Max-pooling operation example.

Hierarchical visual processing

Evidences for coarse-to-fine neural circuits have been found in the early visual cortex (Watt, 1987; Bullier, 2001; Goffaux et al., 2010). Particularly, Bullier (2001) show that magnocellular LGN cells (large receptive field ganglion cell in visual cortex) are poorly selective compared to parvocellular cells. Magnocellular cells transmit rapidly visual information to the parietal cortex and are known to process visual information at a coarse resolution level. The parietal cortex neurons then send back top-down informations in order to inhibit neurons in area V1 and V2, which process visual stimuli at a finer resolution than the precedent circuit. Since the parietal cortex is implied in the dorsal stream visual processing (Mishkin et al., 1983), in visual attention (Behrmann et al., 2004) and in occular saccades control (Bisley and Goldberg, 2003), information retro-injected from this area may modulate V1 neural activity in order to reinforce salient regions and inhibit non-relevant ones in the sense of current attentional target.

From this evidence on coarse-to-fine visual processing in the brain, Brilhault (2014) proposed a model of such interaction based on BCVision. This model used several BCVision-like networks applying learning and recognition at different scales. Networks acting on the lowest resolution also have low resolution version of a pattern stored as a model. For instance, the coarsest network has models with 9x9 pixels size and processes images downsampled by a factor 3. If a model is found at coarse resolution, it is then processed at finer resolution. Brilhault (2014) showed that coarse resolution networks allow more invariance to transformations but at the cost of selectivity. The coarse-to-fine scheme proposed showed that the invariances of coarse network are transfered to the finer-resolution network. It also allows an acceleration of 5 compared with the original single network BCVision version.

2.3.2 Proposed models

To compensate for the previously observed losses in performances in our GPU implementation of BCVision and further increase processing acceleration, we propose to study the influence of the addition of complex cells schemes in BCVision. First, we present a version of BCVision in which we added a layer of complex cells after the gabor-like filtering. Next we present a coarse-to-fine architecture similar to (Brilhault, 2014) proposal which takes advantage of two networks working at different resolutions.

BCVision with complex cells

The first model we propose in this section involves adding a complex cells layer to BCVision. To do so, after the spikes extraction layer, we add a layer which perform pooling on the extracted spike map. It consists in applying a subsampling function, averaging or pooling every region $k \times k$ with an offset s in the spike map. Since spikes are extracted from the binarization of intensities obtained after the filtering step, we simply apply our subsampling step right before binarization. Also, models are learnt from the subsampled spike maps, hence reducing their size by a factor $\frac{32}{s} \times \frac{32}{s}$

The interest of using complex cells here is twofold. First, pooling is known in the literature to add a bit of invariance to transformations while reducing the spatial precision of information transmitted to subsequent layers. In deep architectures, this can be a problem for object detection. But since the current architecture has only one layer of neurons (two if we consider the filtering phase as a neural process), the loss in spatial information is very limited. Secondly, the reason why spatial information is lost during pooling is because this operation reduces each spatial dimension by a factor equal to the stride s. Since the most critical phase in BCVision in terms of processing time is the spike propagation for each model, a prior dimensionality reduction can reduce the computation times by a factor of s^2. Experiments have shown that the best compromise is obtained with $s = 4$, hence reducing the sizes of both spikes maps and models by a factor 16. Since the propagation is convolution based, the theoretical complexity of this algorithm is in $O(W \times H \times C \times C)$, where W, H, C are respectively the width, height and feature number of the spike maps. Since both the models and the input maps are pooled by a factor 4, this results in a theoretical complexity reduction of $4^4 = 256$.

As we mentioned in section 2.3.1, the choice of pooling function had to be benchmarked in order to select the most efficient one between averaging or maximum. A rapid evaluation with the benchmarks described in section 2.2.2 showed that the best performance was reached by using max-pooling instead of average pooling. This results makes sense in the light of literature (Scherer et al., 2010), since the max-pooling operation is best suited for feature-level layers. Also, BCVision relies on the principles of rank order coding, hence the earlier spikes to occur, so it makes sense to extract only the spike for which the activity was maximal for each region.

An illustration of the proposed model is given in Fig. 2.8.

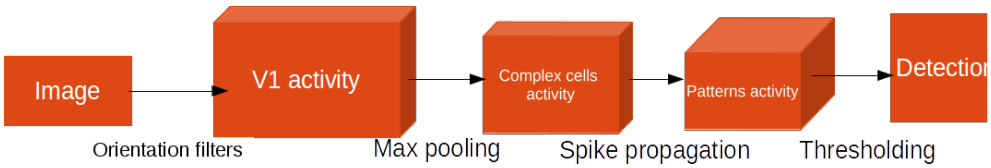


FIGURE 2.8: The proposed BCVision architecture equipped with complex cells

Coarse-to-fine BCVision

Considering the results in terms of performances and processing times obtained by the BCVision coarse-to-fine approach in Brilhault (2014) described in section 2.3.1), the adaptation of such method seemed a good way to compensate the losses of performance observed in the previous Results section and to accelerate further BCVision.

The proposed coarse-to-fine architecture proposed here requires two BCVision networks, one which first detects learnt patterns at a low resolution, and the second which performs full resolution detection only in spatial areas where a detection occurred at low resolution scale. It is expected that the low resolution (coarse) network will have a high recall and raise many false alarms, while the full resolution (fine) network filters increase the overall accuracy.

We propose a variant in the use of subsampling in such coarse-to-fine framework. In the previous work from Brilhault (2014) bilinear subsampling is performed prior to filtering in order to reduce the dimensionality on which computations are performed. We argue that applying bilinear filtering before spike extraction implies a loss in high-frequency information, thus a loss in performances. Moreover since the spikes extraction optimized for GPU architecture has negligible processing time (less than a millisecond for an HD image), we also propose to apply max-pooling operation after the gabor-like filtering step. This allows us to compute the responses to gabor-like just once for both the coarse and the fine network, thus reusing the same map, and to keep the high-frequency information despite the dimensionality reduction.

An experiment to show the difference in performance between a single BCVision performing scaling prior to spikes generation and another single BCVision network using max-pooling after gabor-like filtering is presented in section 2.3.3.

The organization of the different steps is as follows. First, the input image is rescaled with respect to the desired model size to be detected. Next gabor-like filtering is performed. In the coarse network branch, max-pooling is applied to

the gabor responses, followed by the binarization and the spikes extracted this way are propagated with respect to the low-resolution models (learnt from a pooled spike map obtained from the reference image pattern). Following the thresholding a low-resolution detection map is obtained. With the application of a dilatation operation, we obtain a binary mask which is applied to the activity maps (those obtained with the gabor-like filtering step). The fine network can then propagate the masked spikes in order to detect models at full resolution. Note that in order to avoid computations on areas not marked as detections by the coarse network, the fine propagation first checks whether there is a sufficient number of spikes in its input region or not before propagating any model. Since the coarse network generates a mask where only a minor part is marked as a potential detection, the reduction of the number of computations during the fine network propagation step is significant.

The architecture of the coarse-to-fine BCVision is presented in Fig. 2.9.

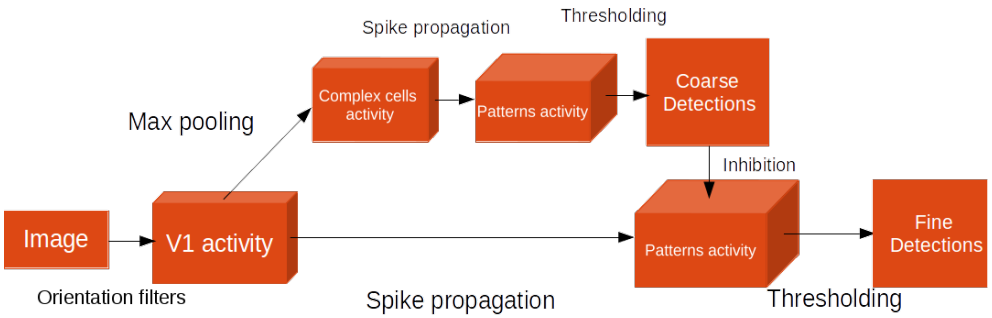


FIGURE 2.9: The proposed coarse-to-fine detection architecture

2.3.3 Experiments and results

In order to evaluate the hypothesis proposed in this section, we used the evaluation datasets already described in 2.2.2.

Firstly we compared the detection performances of our dimensionality reduction method against the one used previously Brilhault (2014). This allows us to determine the best subsampling strategy for the further acceleration of BCVision. Both architecture apply a downsampling factor of 2.

Secondly, we ran the detection performances evaluation on an individual BCVision model with complex cells and on the coarse-to-fine BCVision architecture. Here all the max-pooling operation are performed with a stride of 4 (even if experiments have been done on several downsampling factors).

Finally, the timing evaluation on propagation was run on both the previous architectures.

Every model evaluated in this section was run using a GPU implementation (except for the CPU one). Again, we used for this experiment an Intel i7-3770k for the CPU version and a NVidia GTX 970 for the GPU versions.

Results

Scaling before versus pooling after spikes extraction Fig. 2.10 shows the results obtained running the detection evaluation on a BCVision network applying rescale before spikes extraction and another version where max-pooling is performed after. In almost every condition, the bilinear downsampling prior to the spike map generation shows worst performances than the proposed approach. Both architecture only had equivalent performance level for the scale condition.

The best performance is obtained by the model using complex cells. It goes along with our assumption that applying bilinear interpolation implies loss of high-frequency information, since only the scale condition, where those frequencies are lost by design, shows equivalent performances. The proposed 4approach using max-pooling operation is therefore to be privileged.

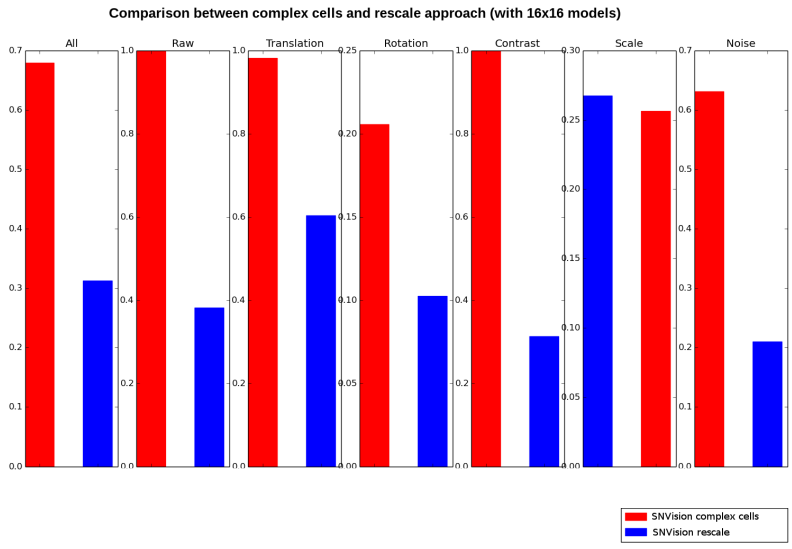


FIGURE 2.10: Detection performances of a coarse subsampled architectures given two subsampling methods, model image rescaling prior to spikes generation versus spike-maps pooling

Detection performance Using the same experimental framework as in the previous section, we studied the performance of the BCVision network with complex cells and the coarse-to-fine architecture. Fig. 2.5 shows different results on the experiments (BCVision software, BCVision Cuda, BCVision with pooling), without the coarse to fine approach. We can observe an overall decrease in F-measure on most transformations as the stride is increased. In more details the drop in performance when applying max-pooling is due to a decrease in precision. Recall is more important in all cases at equivalent thresholds. The decrease in translation performance is explained by the application of max-pooling after the filtering stage. Since only the maximum single pixel activities are retained, translations can lead to sub-sampling different maximums of the model for the same pattern. In the case $s = 2$, the network shows equivalent performances for rotations, contrast and noise conditions whencompared with the GPU version of BCVision without pooling. When $s = 4$, performances are worse in all the conditions except for the images with varying scales. In the latest condition, the use of pooling seems to increase both precision and recall, since the F-Measure is globally greater than with the versions without pooling.

In Fig. 2.11, a comparison of F-Measure performance between high-resolution, low-resolution and coarse-to-fine architectures is presented.

From an overall point of view, the best performance was seen with the coarse-to-fine architecture, the worst being obtained with the single BCVision network with complex cells. Coarse-to-fine architecture outperforms the original BCVision architecture in almost every condition except for the translation and contrast conditions where the loss is low.

The BCVision network equipped with complex cells has the worst performance in every scenario except in the scaling one, which is consistent with the previous section results.

The combination of low and high resolution networks seems to compensate for the inherent drawbacks of both architectures. The high recall of the coarse network allows the fine one to propagate activity to a much lower number of locations. Hence the activity threshold of the fine architecture can be lowered, this network acting as a false alarms filter.

Time performances Using the same experimental framework as before, we studied processing times for low-resolution networks and coarse to fine networks. As before, we recorded the number of models detectable in real time on an HD

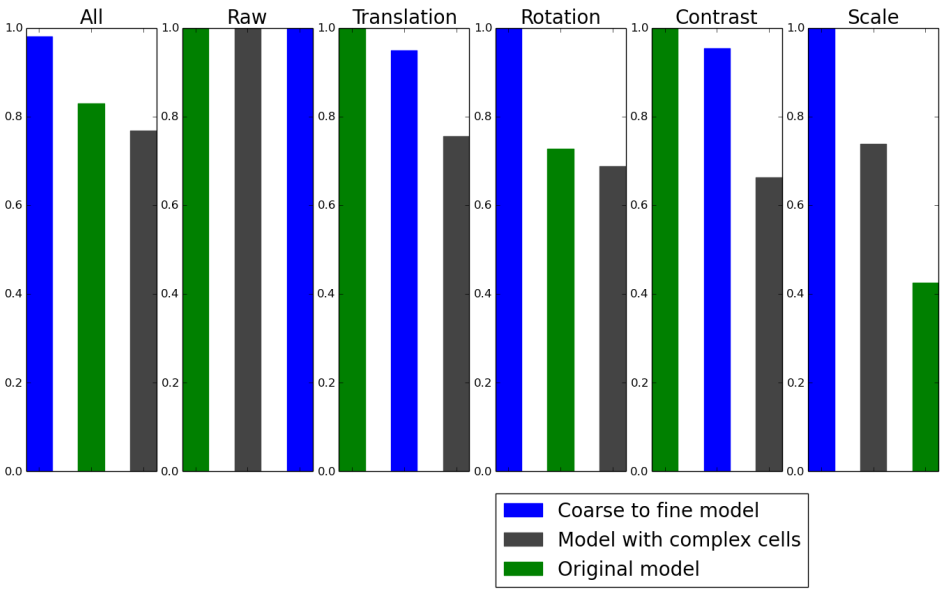


FIGURE 2.11: Detection performances of the base GPU implementation of BCVision, the fully pooled models version and the coarse to fine version. The higher the score in each category, the most robust to the transformation the model.

image at a frame rate of 30 frames per second. Figure 2.12 shows the absolute processing times of the pooling and the coarse-to-fine methods ias a function of the number of models tested (from 1 to 3000).

When the number of models tends towards infinity, i.e. considering only the slopes of the linear regression, the single network with pooling reaches an acceleration factor of 151 when compared to BCVision on GPU, while it reaches a factor of 965 when compared to the CPU version. The same way, the coarse-to-fine method reaches acceleration factors of 143 and 913 compared with GPU version without pooling and the CPU version respectively.

Considering the 30 frames per second real time constraints, the low resolution network can process 3500 models in 30 ms, the coarse to fine method 2100. As a reminder, the CPU version process one model in 101.6 ms, thus 0.3 models in real-time conditions, while the GPU version can process 15 models at 30 frames per second. Hence the coarse-to-fine accelerates the process by a factor 140 compared with GPU version without pooling, and by a factor 7000 compared to the CPU version.

The ratio between the number of models processed in real time by the BCVision architecture with max-pooling and the BCVision architecture without pooling, both on GPU, is equal to 233.33, which is in the order of the theoretical gain

hypothesized in section 2.3.2. However this gain decreases toward 151 when the number of models increases. Although this number is a lower bound of this acceleration factor, our assumption on this decline is a mixed influence of the limitation of computation units for the population count operation, which are four times fewer than most common operations units, and the limitation of memory bandwidth between the devices global memory and the shared memory, which is reached rapidly with the GPU version without pooling and reached later with the version with pooling. Indeed, our theoretical complexity estimation only takes into account the computations, not the memory bandwidth limit, which is often a critical resource on massively parallel hardware optimization.

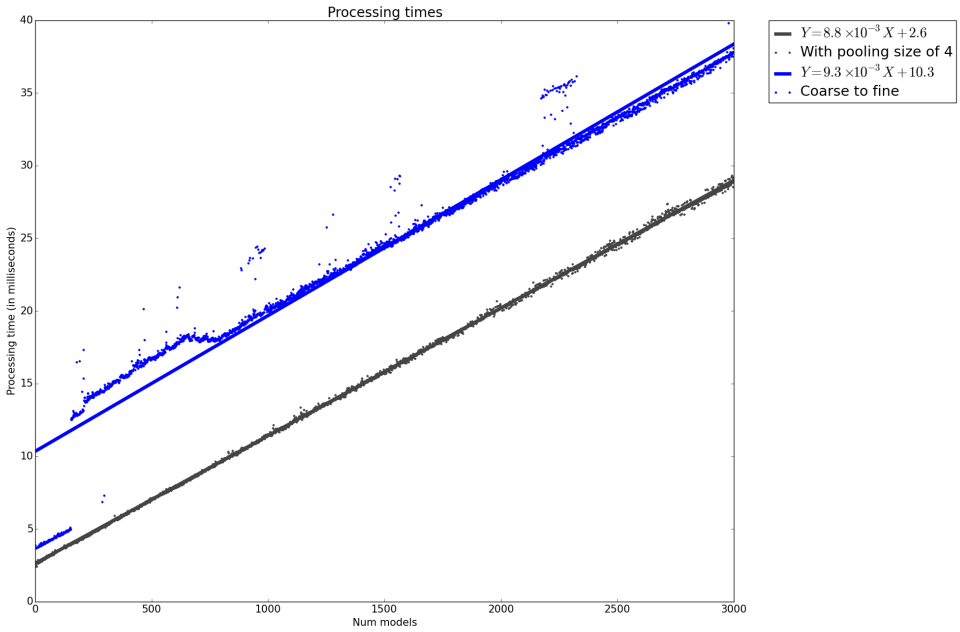


FIGURE 2.12: Acceleration factor in millisecond given the number of models on a single scale 1080p image with a fully pooled architecture version and the coarse-to-fine version of BCVision

2.4 Conclusions

We first propose an algorithm-architecture adequacy study on the implementation of the BCVision visual pattern recognition software on GPU hardware. The different steps of the algorithm were straightforward to port on GPU except for the spike propagation phase which generates the output map of detections. We determined that a threading strategy across output potentials during spikes propagation was the most efficient, converting the iterative sparse list-based propagation of the original model into a convolution problem. We evaluated

performance of this portage in terms of detection performances and processing times. We found that some minor details brought by our optimizations such as the change in the resolution of the models led to a drop in recall, while accelerating the overall process by a factor 6.89.

We proposed to add complex cells, using the max-pooling operation after the spike generation stage led to an overall drop of detection performances, mainly due to a loss in precision as expected. However, this implementation allows an acceleration of the order of one thousand compared with CPU and GPU versions without complex cells. Also we identified during our detection evaluation that this approach can still have a high recall rate.

We finally proposed to combine two networks detecting models at different resolutions. A low-resolution network, which makes use of pooling to detect coarsely the learnt models, allows the masking of the higher-resolution spikes map, which allows a higher-resolution network to focus on fewer spatial location, and thus have its detection threshold lowered to increase its recall. This novel architecture makes it possible to accelerate processing by a factor of one thousand faster than the standard BCVision CPU implementation. Although the networks composing this method are less performing individually than the standard implementation, the hierarchical combination of neural networks increases performance at their initial level.

During the course of this work, we studied possible architectural and algorithmic optimizations to remove technological barriers to BCVision, particularly in terms of processing times. We also made sure that the system did not regress in terms of detection performances.

From a theoretical point of view, the implementation of coarse to fine and the results obtained confirm the results of Brilhault (2014), the multi-resolution approach does not reduce recognition performance while speeding up processing. The differences in our approach lie in the use of complex cells instead of different frequency bands for increase the robustness to the inaccuracies inherent in subsampling algorithms, and in the suitability of our model for massively parallel GPU-like architectures.

Chapter 3

Algorithmic strategies for spike propagation on GPUs

3.1 Introduction

Advances in GPU technology in term of hardware performances and programmability, notably with the improvement of programming languages and SDK like NVidia CUDA, have allowed researchers and engineers to optimize the execution times of neural network computational models.

Some of the first studies proving the efficiency of GPU hardware for neural network optimizations were done in the neuroscience litterature by Brette and Goodman (2012) and in the machine learning community by Krizhevsky (2009). Since then, GPUs have been the hardware of choice for deep neural network training. This reflects the fact that these networks involve many GPU-friendly operations such as matrix dot products, convolutions and global or local reductions.

In the field of neuroscience, the diversity of neuron models has prevented such a consensus. Nevertheless, thanks to the availability of runtime compilation for GPU source code, many research libraries such as BRIAN (Goodman and Brette, 2009) and CarlSim (Beyeler et al., 2015) allowed the use of GPUs to lower the execution times of biological neural network simulations.

Through all the literature in both fields, the majority of the papers related to GPU optimization tends to focus on implementation of neural networks fixing some hyper-parameters[1] while varying others. For instance, Ardakani et al.

[1] Hyper-parameters are variables related to a neural network structure. This naming convention allows these parameters to be separated from those which are optimized during learning.

(2016) studied the ASIC implementation of a single neuron with 1024 inputs with the sparsity variation in the connection matrix. In (Brette and Goodman, 2012), different GPU implementations are tested as a function of the number of neurons in the network in a fully connected manner with fixed sparsity. We observe that the computational complexity is explored only for task or network specific parameters. Different strategies may apply to compute the potentials of neurons in a layer. A first study by Brette and Goodman (2012) used a benchmark of different algorithms for spikes propagation in the context of an overall GPU optimization of the Brian simulator. However, the authors openly admit that the proposed implementations are quite naive and the study of their parameters remains incomplete. Hence no exhaustive computational model is available to know, a priori, the fastest algorithm to use in a given situation.

3.2 Contribution

3.2.1 Aim and hypothesis

The goal of the work presented in this chapter is to design a model of computational efficiency of activity propagation in neural networks, which is the most critical part in the overall inference of such models. We aim to build a model which is able to give an estimate of the computations time for a neural layer. The model parameters includes hardware features like the number of processing cores, the cores clock speed, memory clocks, memory bus size, and layer wise parameters such as the input and output dimensions, input vector sparsity and synaptic matrix sparsity.

In order to focus on the dimensions parameters for this benchmark, we made some simplifications in the neural network model studied. Indeed, we only studied a single layer neural network with only binary activations and synapses, in order to set aside discussions about activations and weight resolution. Also, we focus the study on the computation times of a single feedforward step.

These simplifications allow us to focus on parameters such as :

1. M and P the input and output dimensions of a neural layer,
2. N the number of spikes processed per time step. This parameter encodes the input spike sparsity: the lower N, the higher is the input sparsity.

3. W the number of connections per output neuron. This parameter encodes the sparsity of neurons weights: the lower W, the higher the sparsity is for weights.

We present in this chapter three implementations of spikes propagation on GPU which are compared in terms of computation times. We believe these three implementations generalizes to a full range of methods for spike propagation on synchronous hardware including CPUs, GPUs and FPGAs.

3.2.2 Dense binary propagation

Full resolution dense propagation

The first implementation presented here is a naive implementation, also called dense propagation. It is based on matrix dot product, which is the fundamental building block of many kind of neural networks implementations. Mathematically, it is expressed as follows : given X an input matrix of size (n, M), n being the number of samples (also called batch size in deep learning literature) and M the input dimension, a synaptic kernel K of size (M, P) with P the number of output neuron, and $Y : (n, P)$ the output potential matrix :

$$Y_{i,j} = \sum_{k=1}^{M} X_{i,k}.K_{k,j} \tag{3.1}$$

The single thread sequential implementation and the parallel implementation of such algorithm on GPU are given in 1 and 2 respectively. S is the number

of samples to process in parallel.

Algorithm 1: Sequential dense propagation algorithm with generic resolution

Data: X : input vector (S, M), K : weights matrix (M, P)
Result: Y : output potentials (S, P)

```
1  for s ∈ [0, S[ do
2      for p ∈ [0, P[ do
3          cnt = 0;
4          for i ∈ [0, M] do
5              cnt = cnt + X[s, i].K[i, p];
6          end
7          Y[s, p] = cnt;
8      end
9  end
```

Algorithm 2: Parallel dense propagation algorithm with generic resolution

Data: X : input vector (S, M), K : weights matrix (M, P),
$tid \in [0, max\_threads\_per\_block]$: current thread id, bid : block index, B : the number of thread blocks
Result: Y : output potentials (S, P)

```
1  for s ∈ [0, S] do
2      for b = bid; b < P; b+ = B do
3          cnt = 0;
4          p = b + tid;
5          for i ∈ [0, M] do
6              cnt = cnt + X[s, i].K[i, p];
7          end
8          Y[s, p] = cnt;
9      end
10 end
```

The naive implementation of such algorithm is very efficiently parallelizable on GPU, as shown in a number of studies (Goodman and Brette, 2009; Krizhevsky, 2009; Jia et al., 2014). Indeed, this dot product based implementation can access the global device memory efficiently with coalesced and aligned accesses, avoiding cache faults on the GPU which are a huge performance penalty as shown in 1.3. However, since each neuron potential is computed from all the input and synaptic values, this implementation is also greedy in terms of both

memory and computations.

For instance, a neural layer with inputs and weights at 32-bits resolution with $M = P = 10000$ needs to store the synaptic weights in a memory block of size $10000^2 \times 32 = 400 \times 10^6$ bytes (400MB). This can be highly inefficient when the synaptic kernel has most of its values set to zero.

In terms of computations, this implementation has a theoretical complexity in $O(M.P)$. Ignoring the memory latencies induced by the load and store operations, $M \times P$ MAD (multiply-add) operations must be performed each time the propagation function is called. Following the NVidia CUDA documentation (NVIDIA Corporation, 2010), MAD operation on floating-points or integers have a throughput of 128, meaning that 32 MAD operation can be performed in one clock (considering a warp size of 32 threads).

Binary dense propagation

We now show that binary quantization of neural activations and weights is beneficial in terms of memory transfers and computations.

From a memory point of view, dividing the number of bits necessary to encode an activation or a weight by 32 reduces the memory transfers required between each memory layer by the same factor. Given the same example as the previous section with $M = P = 10000$, the size of the weight matrix in memory can be reduced from 400MB to 12.5 MB.

In terms of computations, dot-products can be implemented with bit-wise logic operation and population count (POPC), the latter counting the number of bits in a given integer or float and available in most GPU systems. As binary values may be interpreted in two ways, bit-wise operations may differ. If zero encodes the real value 0, the bitwise AND between inputs and weights can be used in order to perform the element-wise multiplications part of the dot-product, as in BCVision. Else if zero encodes the value -1 as in Rastegari et al. (2016), bitwise XOR and NOT are equivalent to the element-wise multiplication.

The sequential and parallel implementations of binary dense propagation are respectively given in Algorithm 3 and 4. S is the number of samples to process in parallel. We express X and K as integer-typed matrix, where one integer value is seen as a vector of 32 binary values. Hence, activations and weights are accessed using 32-bit packets in memory. Also, memory ordering matters a lot in order to ensure alignment of memory accesses. The X matrix is stored as S

rows of $M/32$ columns, where the column dimension is coalesced in memory. The matrix K is stored on $M/32$ rows with P columns, hence threads in the same processing block read aligned values. A more rigorous notation would be that K is organized as $(M/32, P, 32)$ bit matrices. We will keep our first notation for simplicity. Such a propagation scheme is also illustrated in Fig. 3.1.

Algorithm 3: Sequential binary dense propagation algorithm

Data: X : input vector $(S, M/32)$, K : weights matrix $(M/32, P)$
Result: Y : output potentials (S, P)

```
for s ∈ [0, S[ do
    for p ∈ [0, P[ do
        cnt = 0; for i ∈ [0, M/32[ do
            cnt = cnt + POPC(X[s, i] AND K[i, p]);
        end
        Y[s, p] = cnt;
    end
end
```

Algorithm 4: Binary dense propagation algorithm

Data: X : input vector $(S, M/32)$, K : weights matrix $(M/32, P)$, $tid \in [0, max\_threads\_per\_block]$: current thread id, bid : block index, B : the number of thread blocks
Result: Y : output potentials (S, P)

```
for s ∈ [0, S] do
    for b = bid; b < P; b+ = B do
        cnt = 0;
        p = b + tid;
        for i ∈ [0, M/32] do
            cnt = cnt + POPC(X[s, i] AND K[i, p]);
        end
        Y[s, p] = cnt;
    end
end
```

Theoretical processing times for binary dense propagation

As shown in algorithm 4, the number of operations required for one instance of the function is only dependent on parameters M and P. Thus, a computational

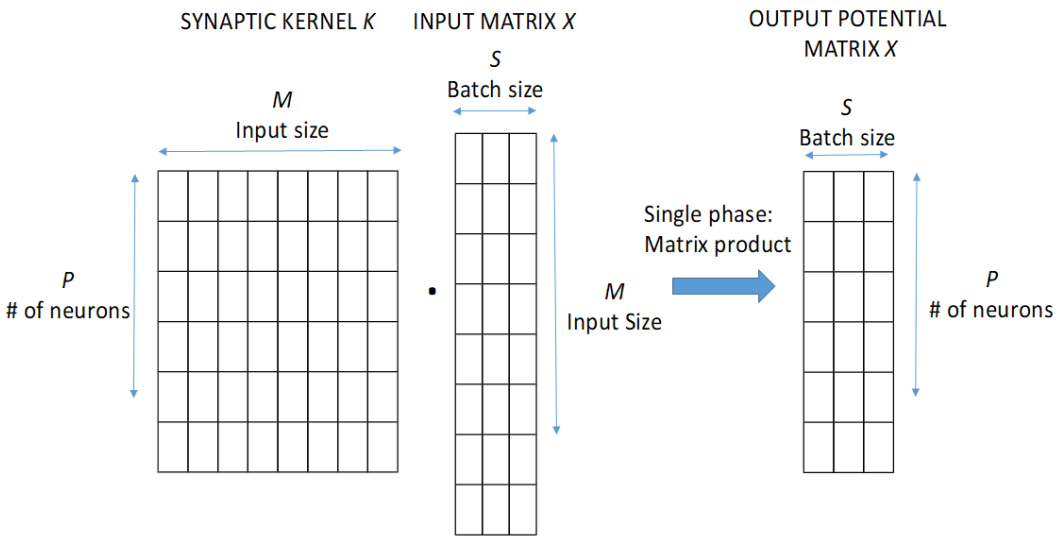


FIGURE 3.1: Binary dense propagation algorithm illustration

model which predicts the processing times given the parameters is straightforward to design.

Let us define the different parameters of the computational model for a generic hardware implementation.

First we define the hardware parameters.

- SM is the number of parallel blocks which can be run concurrently (known as Multi-Processors on CUDA)
- F_{cl} is the frequency in MHz of the processing cores.
- T_{AND},T_{POPC} and T_{ADD} are the theoretical throughputs of the bitwise-AND, POPC and integer ADD functions respectively. Theoretical throughput is directly linked to the number of dedicated compute-units on each multi-processor for a given operation, or whether an operation is translated into multiple instructions at compile-time. Note that we do not take into account the total number of processing cores on the device, since the number of multi-processors with the throughput of each operation are more precise for our model and already contains this information. For instance, a NVidia GTX Titan X Maxwell has 3072 total CUDA cores distributed over 24 multi-processors, or 128 CUDA cores per SM, which is also the theoretical throughput of the most optimized operations units.
- F_{mem} is the memory frequency
- B is the transfer bus size

- Ca,Ra and Wa are the column access latency, the row access latency and the write access latency respectively. Note that Ra must be interpreted as a row-access overhead, hence when an access to a different row is commanded by the device, the total latency to the memory space is equal to $Ca + Ra$.

For a given algorithm, we can estimate the number of core clocks C_{cl} required to perform the computations, the number of aligned read Acc_r, aligned writes Acc_w and cache faults Acc_f induced by the algorithm. In addition, one must define the number of threads N_{th} launched on each multi-processor. On CUDA, the number of threads ranges between 32 and 1024. In the binary dense case:

$$C_{cl} = \frac{M.P}{32}.(\frac{1}{T_{AND}} + \frac{1}{T_{POPC}} + \frac{1}{T_{ADD}}) \tag{3.2}$$

$$Acc_r = M.(1 + P) \tag{3.3}$$

$$Acc_w = P \tag{3.4}$$

$$Acc_f = \frac{M.P}{32.N_{th}} \tag{3.5}$$

Having defined these metrics from algorithm 4, we are now able to compute the processing times given the parameters. We note T_m the memory transfer times, T_{cr} the core computations time and T_{tot} the total processing time for one step of propagation. We also define O_k an eventual overhead due to function call or synchronization between the device and the host.

$$T_{cr} = \frac{C_{cl}}{F_{cl}.SM} \tag{3.6}$$

$$T_m = \frac{1}{F_{mem}.B}.(Ca.Acc_r + Ra.Acc_f + Wa.Acc_w) \tag{3.7}$$

$$T_{tot} = T_{cr} + T_m + O_k \tag{3.8}$$

3.2.3 Sparse binary propagation

While the implementation on GPU of the dense dot-product for spikes propagation allows a great acceleration in light of the literature, this algorithm computes all the correlations between synaptic weights and input spaces in order to get the potential of output neurons. In the case of sparsity, when a lot of values (inputs or weights) are zeros, this implementation obviously computes many unrelevant spikes and weights since these values does not carry any useful information to propagate.

We propose here an approach for propagation of sparse input spikes across a layer. We define the notion of subpackets of spikes, a subpart of the input space indexed by its relative position in it. A subpacket is first defined by its size S_p. We choose this size according to the lowest resolution for which a given hardware is optimized. For instance, consumer-grade NVidia GPUs are optimized for computations on 32-bit data, so a subpacket can be implemented on an integer value. Next, the subpacket is referenced in memory as an element of a matrix I which contains the addresses of X where a subpacket contains at least one spike. Note that if the input space is of size M, the elements of I will be defined on the interval $(0, \frac{M}{S_p})$.

In order to apply the sparse binary propagation, we start from a input spikes vector X and a synaptic kernel K in the same format as in the dense format. The first step of this algorithm is the construction of the array I given X. This can be performed by counting the bits of each subpacket in X and appending the index to I if the result is greater than zero. The number of subpackets $N_s \in [0; \frac{M}{32}]$ generated this way depends both on the number of spikes N in X, and the distribution of the input spikes. For instance if input spikes tend to be spatially correlated, it is more likely that two spikes belong to the same subpacket, hence reducing N_s. In contrast, if spikes tend to be uniformly distributed across all the input space, N_s may increase rapidly as the number of spikes N increases. An illustration of the proposed method is shown in Fig. 3.4.

The uniform distribution of spikes being a worst-case scenario, it is worth studying the behaviour of the algorithm with such distribution in order to estimate a upper bound of the processing times. We first determine $P(X_i = 0)$, the probability that no spike is present at location i in the input space X. Given N the number of distinct non-zero spikes in X and M the size of X, we have :

$$P(X_i = 0) = 1 - \frac{N}{M} \tag{3.9}$$

The probability that no spikes among the N belongs to a subpacket J of size S_p is then given by :

$$P(X_J \quad empty) = \prod_{i \in J} P(X_i = 0) = (1 - \frac{N}{M})^{S_p} \tag{3.10}$$

Fig. 3.2 shows the distribution of such probability in function of the parameters M and N.

As the number of subpackets is equal to $\frac{M}{S_p}$, the average number of subpacket which contains at least one spike (hence have to be processed) can be estimated by the following equation :

$$N_s = \frac{M}{S_p}.P(X_J \quad empty) = \frac{M}{S_p}.(1 - \frac{N}{M})^{S_p} \tag{3.11}$$

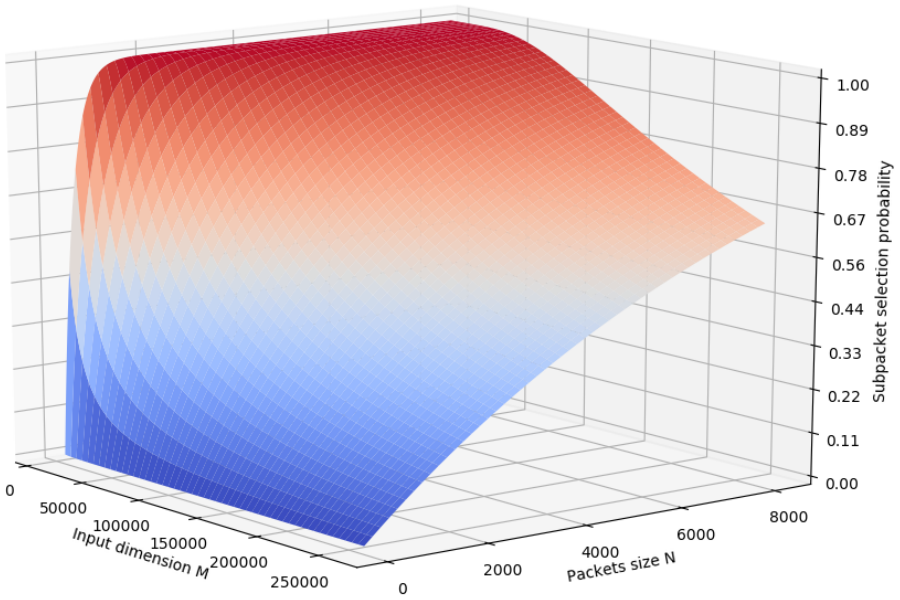


FIGURE 3.2: Probability for a subpacket of size $S_p = 32$ to contain at least one spike, as a function of M and N.

Once the two lists have been constructed, the propagation is performed as in Algorithm 5. The parallel version is also shown in Algorithm 6.

Algorithm 5: Sequential binary sparse propagation algorithm

Data: X : input vector $(S, M/32)$, I : input indexes (S, N_s) , K : weights matrix $(M/32, P)$
Result: Y : output potentials (S, P)

```
for S ∈ [0, S[ do
    for p ∈ [0, P[ do
        cnt = 0;
        for i ∈ [0, N_s[ do
            cnt = cnt + POPC(X[s, I[s, i]] AND K[i, I[s, i]]);
        end
        Y[s, p] = cnt;
    end
end
```

Algorithm 6: Parallel binary sparse propagation algorithm

Data: X : input vector (S, M), I : input indexes (S, N_s) , K : weights matrix $(M/32, P)$, $tid \in [0, max\_threads\_per\_block]$: current thread id, bid : block index, B : the number of thread blocks
Result: Y : output potentials (S, P)

```
for s ∈ [0, S] do
    for b = bid; b < P; b+ = B do
        cnt = 0;
        p = b + tid;
        for i ∈ [0, N_s[ do
            cnt = cnt + POPC(X[s, I[s, i]] AND K[i, I[s, i]]);
        end
        Y[s, p] = cnt;
    end
end
```

Theoretical processing times for binary sparse propagation

Having defined the hardware parameters in section 10, we propose now a computational model for the binary sparse algorithm. Equations from 3.6 to 3.8 remain the same as before. Only equations from 3.2 to 3.5 are modified accordingly to algorithm 6.

$$C_{cl} = \frac{N_s.P}{32}.(\frac{1}{T_{AND}} + \frac{1}{T_{POPC}} + \frac{1}{T_{ADD}}) + \frac{M}{32.T_{POPC}} \tag{3.12}$$

$$Acc_r = 32.N_s.(2 + P) + \frac{M}{32} \tag{3.13}$$

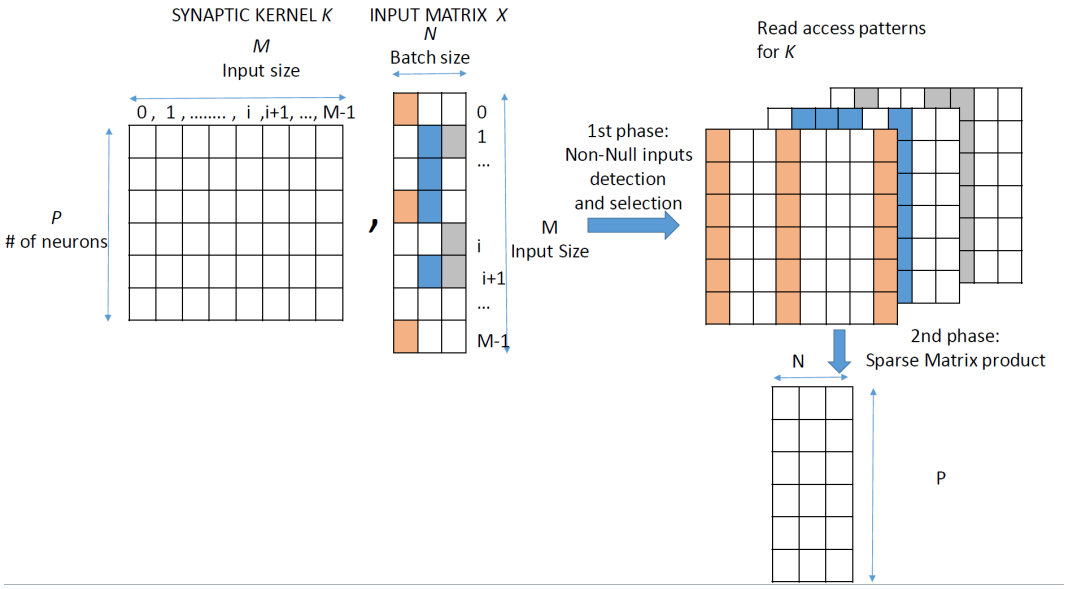


FIGURE 3.3: Binary sparse propagation algorithm illustration

$$Acc_w = 32.P + N_S \tag{3.14}$$

$$Acc_f = 3.N.P/SM \tag{3.15}$$

3.2.4 Output index histogram

The last algorithm we propose, based on an original idea of Jacob G. Martin[2], also aims to reduce the computations of non-informative values such as zeros. If a network shows high sparsity in the input or the weight space, the previous two approaches still perform computations over zeros, which do not influence the result of the potential in this binary scheme.

We present a third algorithm which performs fast propagation of binary activity in very sparse neural networks. In this algorithm, connections are stored as multiple lists of indexes instead of a dense description (with MxP elements). Each input is associated to the list of the output neurons it is connected to. Inputs are stored as a list of indexes instead of a full vector with zeros and ones. In both inputs and weights, indexes are only kept in memory if the connection actually exists, in other words where a one would be used in a dense format. We note K_L the connection matrix, which stores for each row in the range $[0, M[$

[2]Jacob G. Martin is post-doc in MAOS team at CNRS CerCo

a connection list containing the addresses of all the output neurons an input neuron is connected to.

When a spike packet is presented, a propagation index list P_L is built by concatenating the connection lists referenced in the input packet. The propagation list P_L contains the indexes of the output neurons whose potentials must be incremented, each index can be present multiple times in this list. When the propagation list is built, a histogram kernel with P bins is run on this list. The resulting histogram contains the potentials of all the output neurons.

Starting with an input index list I of size N, the first function presented in algorithm 7 in its sequential version gathers the connection lists indexed by I from the input-to-output matrix K_L. This results in a propagation list P_L which contains all the outputs to update. We consider here every connection list in the weight matrix K_L as a variable size list k_n.

Algorithm 7: Sequential construction of the propagation list

Data: I : input indexes (S, N), K_L : weights matrix (M, k_n)
Result: P_L : propagation list $(S, k_{max}.N)$

```
1 P_L = EmptyList();
2 for s ∈ [0, S[ do
3 |   s_list = EmptyList();
4 |   for n ∈ [0, N[ do
5 |   |   s_list.Append(K_L[n, i]);
6 |   end
7 |   P_L.Append(s_list);
8 end
```

In order to implement in a parallel fashion this algorithm, we must, for each thread, read a selected value and copying it in P_L at a globally shared index. This global index is incremented concurrently through the atomic ADD instruction, which implies a locked access to this variable. This may cause large latencies while waiting for synchronizations and unlocking on the global index variable. As the use of atomic global operations is suboptimal, we propose another strategy in order to build the propagation list.

Since we want to avoid misaligned data during reading of the connection lists, we now store all the input-to-output connections in a matrix K_L of size (M, k_{max}), where k_{max} is the maximum list length across every input-to-output list. Every row is filled with the input-to-output indexes of a given input index, and if the list is shorter than k_{max}, a filling value (for instance -1) is padded on

the remaining values. Note that in practice, k_{max} should ensure 256-bits data alignment in order to avoid memory access overheads while reading the device memory. Hence, k_{max} may be replaced by the closest integer greater than k_{max} that can be divided by 256 over the index resolution. In our case, the resolution being 32-bits, $k_{max} := ((k_{max}/8) + 1) \times 8$ with / operation being the integer division.

The second approach to implement the propagation list construction starts with the allocation of P_L as a $(N, k_{max}))$ matrix filled with an impossible index value (i.e. -1). Basically, a first kernel simply copies the selected lists sequentially into P_L. In order to discard all the impossible index values, a second kernel performs a selection algorithm on all the propagation list values, giving a filtered P_L matrix of size k_{list}. Basically, the selection algorithm has each block of threads read a row in P_L, and performs an atomic global operation only to get its offset. Once the offset has been obtained by the block, selected values are stores sequentially. We observed this second approach, avoiding the use of the global atomic ADD for each value, was three times faster than the first one. Algorithm 8 presents the first kernel of this approach. The second kernel has been implemented with the DeviceSelect function from the CUB NVlabs library [3].

Once the propagation list have been built, we may now proceed to the propagation phase shown in algorithm 9. Basically, this kernel counts the number of occurrences of each output in the propagation list, resulting in the final potential vector. In algorithm 9, threads are organized in blocks to perform counting. A count vector is stored in the shared memory for a block of thread. The size of this vector is constrained by the physical hardware memory size of shared memory, which is 48 kb by default on CUDA devices since the introduction of the Maxwell architecture. Hence each thread block computes the potential of the output neurons between P_i and P_j, where $i - j$ equals the shared memory size divided by the resolution of the output values. Let us call this interval S_H. Each thread reads all the values in the propagation list P_L with index equals to its id modulo the block size. If an input index value is in the considered output indexes range, the thread will perform an atomic ADD in shared memory for this output neuron. Atomic ADDs in shared memory are significantly faster than its equivalent in device memory. When all the propagation list have been entirely read, output potentials are written in the device memory and the block terminates.

[3] https://nvlabs.github.io/cub/index.html

The choice for launch configuration, in other words the number of thread blocks and the number of threads per block, is hardware specific here. The larger the number of threads per block, the better the memory caching will be. Hence we can choose in our case 1024 which is the maximum number of threads per blocks on a Maxwell GPU. In algo. 8, the number of blocks can be the number of input spikes, hence each block is responsible for the copy of one row. A more efficient global loop scheme can also be implemented, hence setting the number of blocks to the number of multi-processors on the device is a simple and efficient choice. In algo. 9, we also set the number of threads to the maximum. The number of blocks can be either $\frac{P}{S_H}$ if each block is responsible for updating S_H, or to the number of streaming multi-processors if the CUDA kernel is implemented with a global loop scheme.

A summary of this algorithm is shown in Fig. 3.4.

Algorithm 8: Construction of the propagation list

Data: I : input indexes (S, N), K_L : weights matrix (M, k_{max}), $tid \in [0, max\_threads\_per\_block]$: current thread id, bid : block index, B : the number of thread blocks

Result: P_L : propagation list $(S, k_{max}.N)$

```
for s ∈ [0, S] do
    for b = bid; b < N; b+ = B do
        for t = tid; t < k_max; t+ = max_threads_per_block do
            P_L[s, b × k_max + t] = K_L[I[s, b], t];
        end
    end
end
```

Algorithm 9: Histogram propagation algorithm

Data: P_L : filtered propagation list (S, k_{list}), S_H : the block processing interval, $tid \in [0, max\_threads\_per\_block]$: current thread id, bid : block index, B : the number of thread blocks

Result: Y : output potentials (S, P)

```
sharedCnt = shared_array[S_H];
for s ∈ [0, S] do
    for b = bid; b < N; b+ = B do
        low = b × S_H;
        high = (b + 1) × S_H;
        for t = tid; t < k_list; t+ = max_threads_per_block do
            if low ≤ P_L[s, t] ≤ high then
                id = P_L − low;
                atomicAdd(sharedCnt[id], 1);
            end
        end
        for t = tid; t < S_H; t+ = max_threads_per_block do
            Y[s, low + t] = sharedCnt[t];
        end
    end
end
```

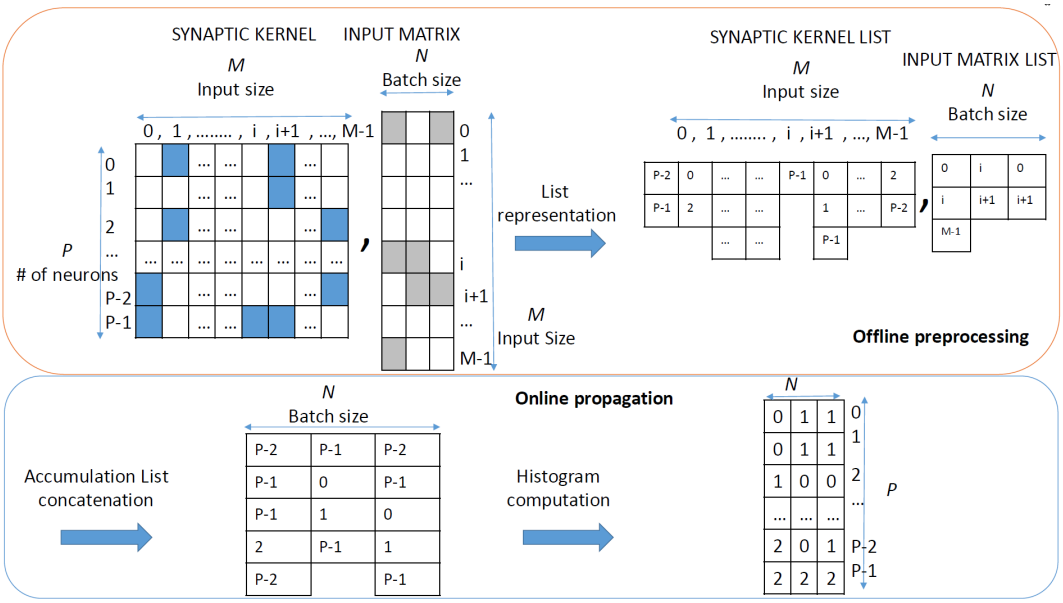


FIGURE 3.4: Binary histogram propagation algorithm illustration

Theoretical processing times of binary sparse propagation

Taking the same hardware parameters as in section 10, the computational model for the histogram algorithm we propose is as follows. This computational model combines both algo. 8 and 9. We ignore here the selection phase since the precise implementation is unknown to us, and it appears in simulations to be negligible. We consider that the weight matrix is filled following a uniform distribution of $P \times W$ ones over $M \times P$ slots. Each input is then connected on average to $\frac{P \times W}{M}$ outputs. We also introduce T_{COMP} the throughput for integer comparison and $T_{ATOMICADD}$ the throughput for a shared atomic add.

$$C_{cl} = N \times \frac{P \times W}{M}\left(\frac{1}{T_{ADD}} + 2.\frac{\frac{P}{S_H}}{T_{COMP}} + \frac{\frac{P}{S_H}}{T_{ATOMICADD}}\right) \tag{3.16}$$

$$Acc_r = N.\frac{P \times W}{M}.(1 + 32 \times ceil(\frac{P-1}{S_H} + 1) \tag{3.17}$$

$$Acc_w = 32.(P + N) \tag{3.18}$$

$$Acc_f = N \tag{3.19}$$

3.3 Experiments and results

3.3.1 Material and methods

We want to model deterministically the behaviour of the three defined algorithms given their parameters M, P, N and W. In order to reduce the exploration space, we first set for the benchmark phase : $M = P$. The equality between these two parameters allows us to simplify the parameter fitting during our benchmarking and facilitates the interpretation of data.

We run a benchmark on a range of parameters value in order to output the CUDA kernel processing times, which will help in the prediction of the most efficient algorithm given task specific parameters. It will also allow us to confirm the computational model we presented for each of the proposed algorithm.

We also compare the efficiency of the three proposed methods. To do so, we define a spike throughput metric (i.e. the number of input spikes processed per second), basically obtained by dividing N by the kernel processing time in seconds. This metric is relevant as both binary dense and sparse methods perform computations over zeros, which are wasted computations. Hence this last experiment will help us determine, given the layer parameters, whether performing computations on zeros anyway is more efficient or not.

We chose the following ranges for the different parameters.

- Input / output size M : 1024 to 65536 with multiplicative step of 2
- Number of spike per packet N : 1 to 4096 (with $N < M$) with multiplicative steps of 2
- Connections per output neuron W : 1 to 4096 (with $N < M$) with multiplicative steps of 2

All the implementations were done for a single sample per propagation such as $S = 1$. Evaluations were made in order to match the worst case scenario for each implementation. Since the dense algorithm is agnostic to parameters N and W, any set of parameters or spikes distribution in the input space can be considered best and worst cases. For the sparse algorithm, its efficiency relies on the spatial correlation between input spikes. Hence having spikes to follow a uniform distribution with probability $\frac{N}{M}$ leads to the worst case scenario. In practice, indexes from 0 to M are randomly shuffled and the first N elements in

this list are sent for propagation. For the histogram algorithm, the worst cases emerge directly from the parameter tuning as shown in the Results section.

We run our experiments on an overclocked NVidia GTX Titan X Maxwell GPU. For such device, we can determine the hardware parameters of our model as follows:

- $F_{cl} = 1.4 \times 10^9$
- $SM = 24$
- $B = 384$
- $F_{mem} = 7.9 \times 10^9$

However, we found no information about CAS, RAS and write latencies (resp. Ca, Ra, Wa) for any NVidia device. Moreover, while we explicitly avoided any memory caching effect as much as possible, specifically by declaring inputs as volatile, we were not able to completely annihilate them. In this sense, we chose to estimate them with least-square parametric optimization with respect to all the timings obtained with the three proposed methods. We found for such device the following parameters:

- $Ca = 1.7947$
- $Ra = 0.6137$
- $Wa = 22.5326$
- $O_k = 14.4593$

Considering the memory-caches influences, the different values obtained this way are consistent. For instance, memory-write accesses are far more slower than read access of a order of magnitude of 10, and the kernel overhead O_k is perfectly in the order of magnitudes of the kernel launch overheads.

3.3.2 Kernel times analysis

In this section we show how the processing times obtained by each algorithm vary as a function of the network parameters.

Binary dense method

Fig. 3.5 show the timings obtained with our benchmark varying the different parameters M, N and W, along with a surface showing the estimated kernel time processing with our computational value.

We first observe that, in line with our computational model, kernel processing times only depends on the variation of parameter M. As show in two dimensions in Fig. 3.6, the behaviour of this method is quadratic in function of the input and ouput dimensions of the layer. The difference between the real data points and the estimated surface directly emerges from the error of optimization due to the complex estimation with the remaining caching mechanisms.

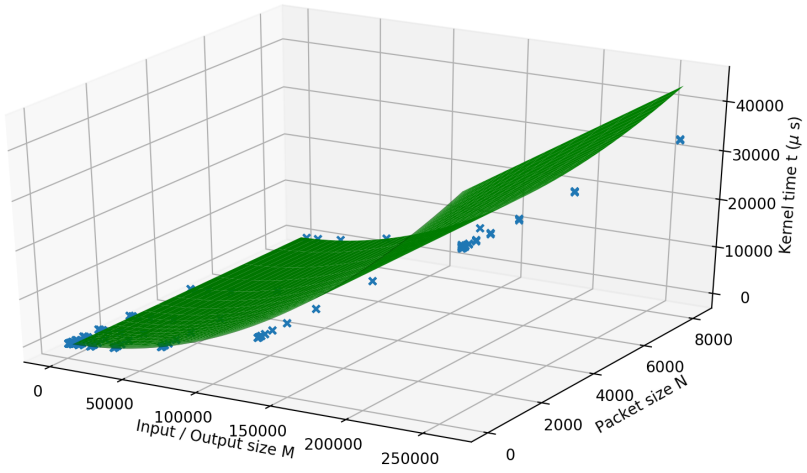


FIGURE 3.5: Processing times of the binary dense algorithm in function of M and N. In blue, real data point, in green the computational model estimation

Binary sparse method

For this method kernel processing times can be estimated by the product between the computational model of binary dense method (Fig. 3.5) and the probability distribution that a subpacket contains a spikes (Fig. 3.2) under our worst-case hypothesis of a uniform distribution of spikes in the input space. To be exact, and as we will demonstrate later, an additional term including the access overhead to both X and I (instead of X only in the previous method) must be taken into account. Fig. 3.7 presents the obtained kernel times and the estimated computational times as a function of M and N.

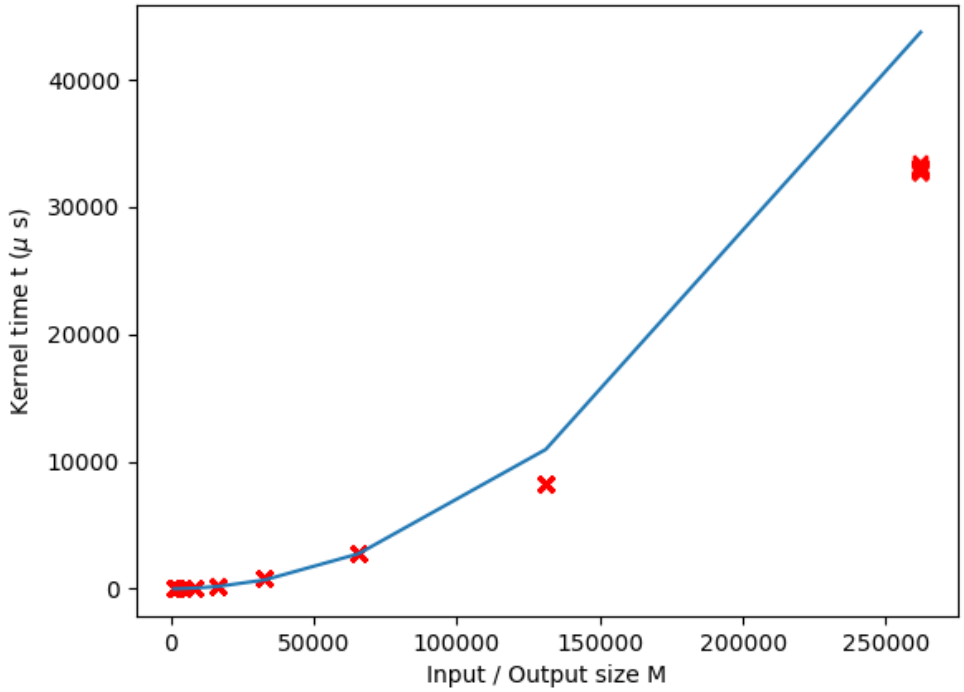


FIGURE 3.6: Processing times of the binary dense algorithm in function of M. In red, real data point, in blue the computational model estimation

Fig. 3.8 shows the same data but slicing the surface across the dimension M. Here we can see more precisely on this graph the effect of N on the processing times, which follows the negative power function described by the subpackets distribution. In Fig. 3.9, slicing is performed over the N dimension, showing the same quadratic behaviour as the binary dense method. As expected, W has no influence on processing times with this method.

Output index histogram method

Fig. 3.10 shows the processing times as a function of W following different values for N. We can clearly see that processing times follow a linear behaviour in function of W. Looking at Fig. 3.11, processing times are also linear as a function of N.

We can also notice that for relatively large values of W and N (8192 on the figures), the data points begin to spread, indicating a slight influence of M on

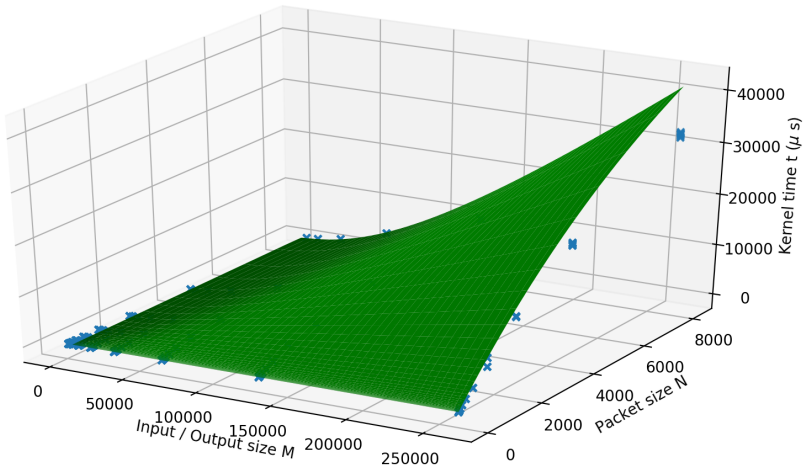


FIGURE 3.7: Processing times of the binary sparse algorithm as a function of M and N. In blue, real data point, in green the computational model estimation

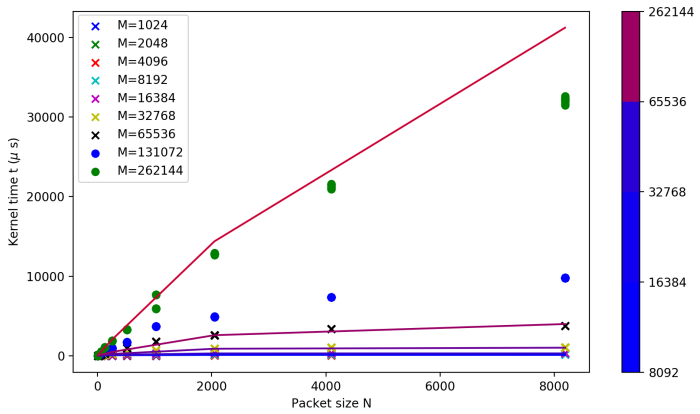


FIGURE 3.8: Processing times of the binary sparse algorithm as a function of N. The different sets of points represent real data points, while the curves represent the computational model estimation. The colorbar maps the colors of the different curves to the corresponding value of M.

the processing times. Indeed as M increases, the number of computations required to build the propagation list also increases, resulting in a small overhead function of M.

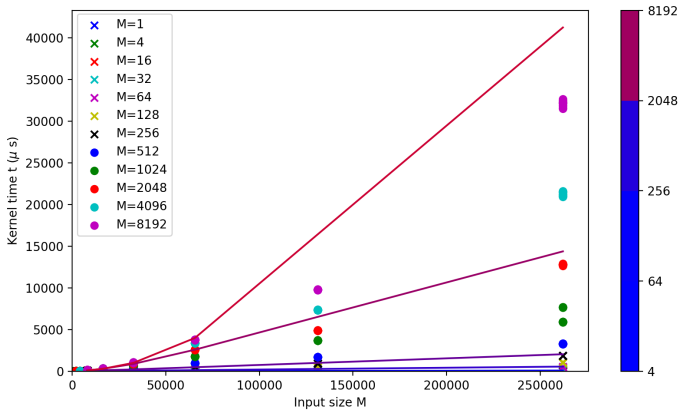


FIGURE 3.9: Processing times of the binary sparse algorithm as a function of M. The different sets of points represent real data point, while the curves represent the computational model estimation. The colorbar maps the colors of the different curves to the corresponding value of N.

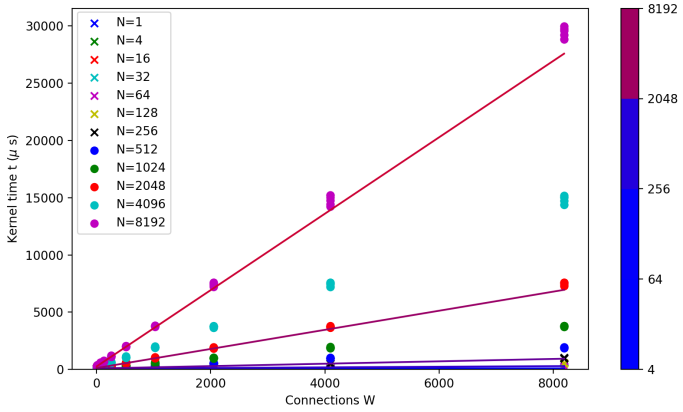


FIGURE 3.10: Processing times of the output indexes histogram algorithm as a function of W. The different sets of points represent real data point, while the curves represent the computational model estimation. The colorbar maps the colors of the different curves to the corresponding value of N.

3.3.3 Spikes throughput analysis

We now analyse the spike throughput of each proposed method and then identify their range of efficiency given the parameters. Fig. 3.13 shows the input spike throughput in log10 scale for the three methods given parameter M, N, and W. A first general observation is that each method has effectively a range of parameters where it performs better than the others.

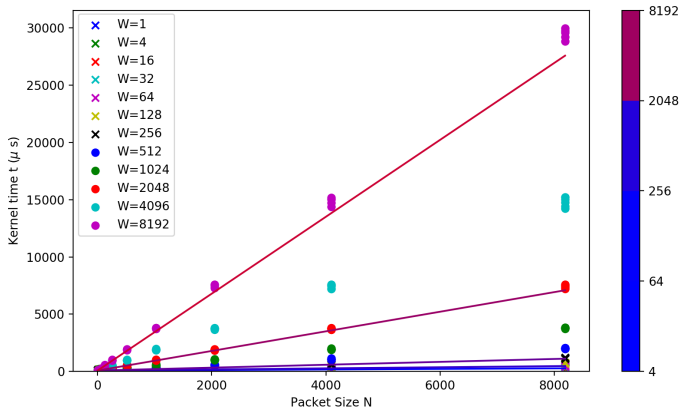


FIGURE 3.11: Processing times of the output indexes histogram algorithm as a function of N. The different sets of points represent real data point, while the curves represent the computational model estimation. The colorbar maps the colors of the different curves to the corresponding value of W.

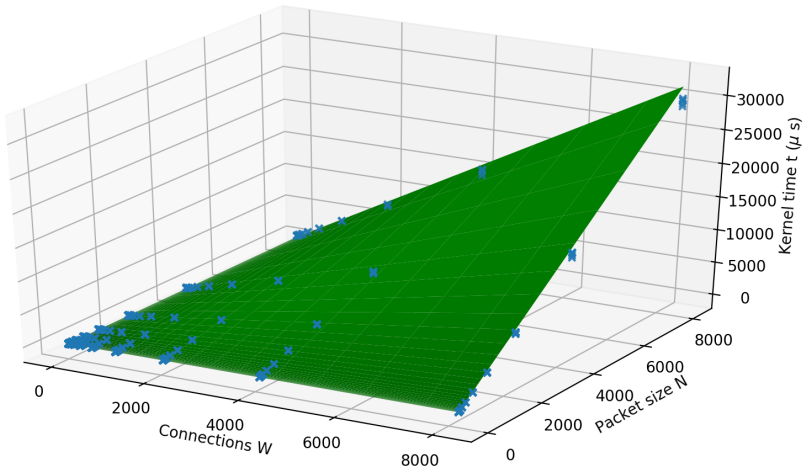


FIGURE 3.12: Processing times of the output indexes histogram algorithm as a function of W and N. In blue, real data point, in green the computational model estimation

The binary dense method is the most efficient when N is high, while performing poorly when N is low. Indeed as the sparsity in input and weight spaces increases, the number of wasted computations involving zeros also increases.

The binary sparse method outperforms the dense approach when the sparsity in the input space is high. It also performs better than the output indexes histogram method as the sparsity in the weight space is reduced.

Finally, performance of the output indexes histogram method decays rapidly as N and W increase. However its quasi-independence to the input and output space sizes M makes it really efficient for cases where these dimensions are high and parameters N and W are low.

3.4 Conclusions

In this chapter we have studied three models of spike propagation (namely dense, sparse and histogram), and give a explicit complexity formula. We have experimented and explored according to 3 parameters: M,N, and W. Such a study allowed us to obtain a computational model of the processing time given these parameters for each method. We are now able to analyse the range of parameters for which each method is the most efficient.

3.4.1 When to use a given approach?

The binary dense method is sparsity agnostic. While this feature is penalizing in the case of layers with high sparsity, this naive approach remains easily predictable through its quadratic behaviour over input and output sizes. This approach is best suited for many neural networks in the machine learning literature, since the efforts to reduce sparsity through learning (Liu et al., 2015; Wen et al., 2016) or with pruning methods (Molchanov et al., 2016) are not the priority of this domain. Also, except when there is a classification loss function (for instance softmax), fully-connected layers rarely have large input and output sizes. Indeed, AlexNet (Krizhevsky, 2009) is a model with one of the largest fully-connected layers with 4096 neurons with low sparsity, hence in the perfect range of efficiency of the dense approach.

The binary sparse approach depends on both the input and output dimensions of the layer and the sparsity over the input space. Since it shares its behaviour regarding the layers dimensions with the binary dense method, its relative efficiency compared to this latter method depends only on the sparsity of the input space and on the size of the subpackets S_p. Fig. 3.14 shows the probability that a subpacket contains at least one spike as a function of the ratio $\frac{N}{M}$ and the subpacket size S_p in the worst-case hypothesis of a uniform distribution of spikes over the input space. We can notice that the larger the packet size, the higher must be the sparsity to be more efficient than the binary dense method.

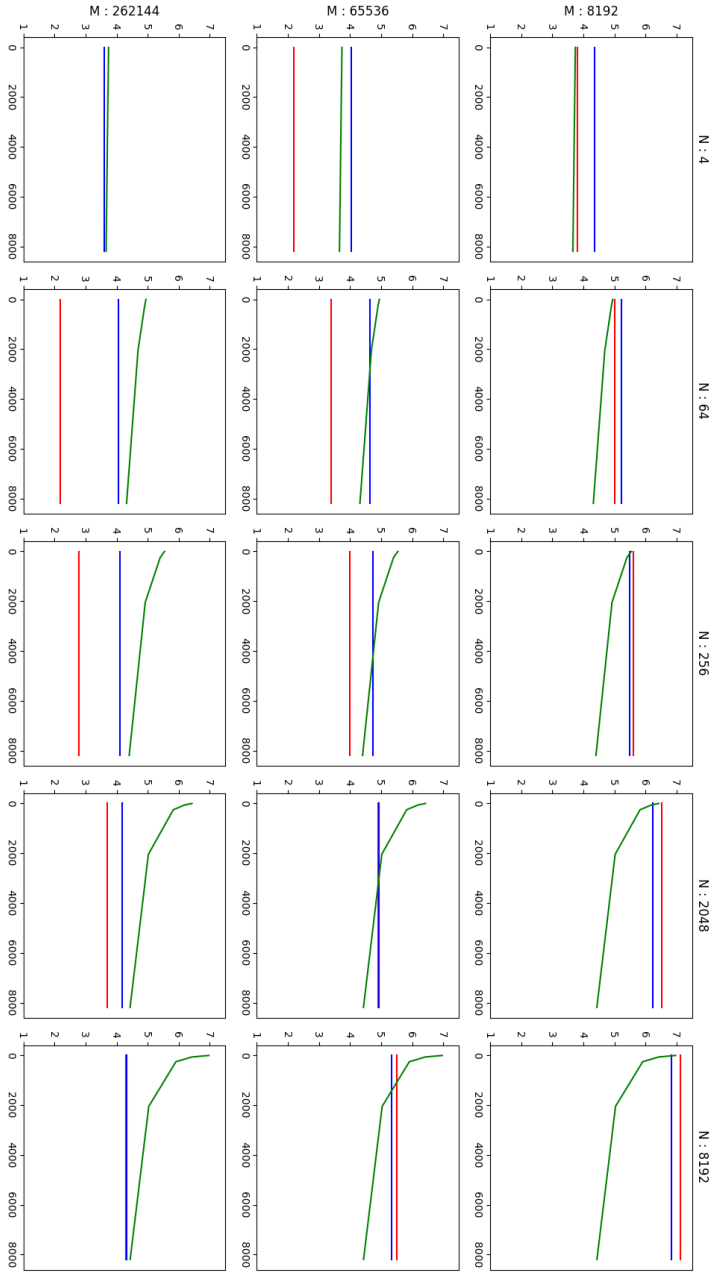


FIGURE 3.13: Spike throughput comparisons between the three proposed methods. In red, the binary dense method, in blue the binary sparse method and in green the output indexes histogram method are shown. Each graph in a row corresponds to a single value of M, while each graph in a column corresponds to a single value of N. For each graph, the parameter W is represented on the x-axis, the spike throughput is represented on the y-axis. Note the spike throughput is represented using a **log 10 scale**.

Note also the case $S_p = 1$, where we can retrieve the same linear behaviour regarding the input sparsity. For our experimental value of $S_p = 32$, the sparse approach performs best for a ratio $\frac{M}{N} \leq 0.2$, so a corresponding sparsity of 0.8. As neural networks compression techniques (Han et al., 2015; Molchanov et al., 2016) allow the sparsity to reach 0.9 with almost no accuracy loss, our sparse method can be relevant to optimize such compressed / pruned networks. Moreover, reducing the subpacket size S_p lowers the sparsity requirement to reach better performances than with naive methods. While the absence of bit counting ALUs for lower precision than 32-bits on GPUs makes it difficult to lower the subpacket size, it is still perfectly practicable on more specialized hardware such as FPGAs (as it has already be done in Thorpe et al. (2017)) and ASICs.

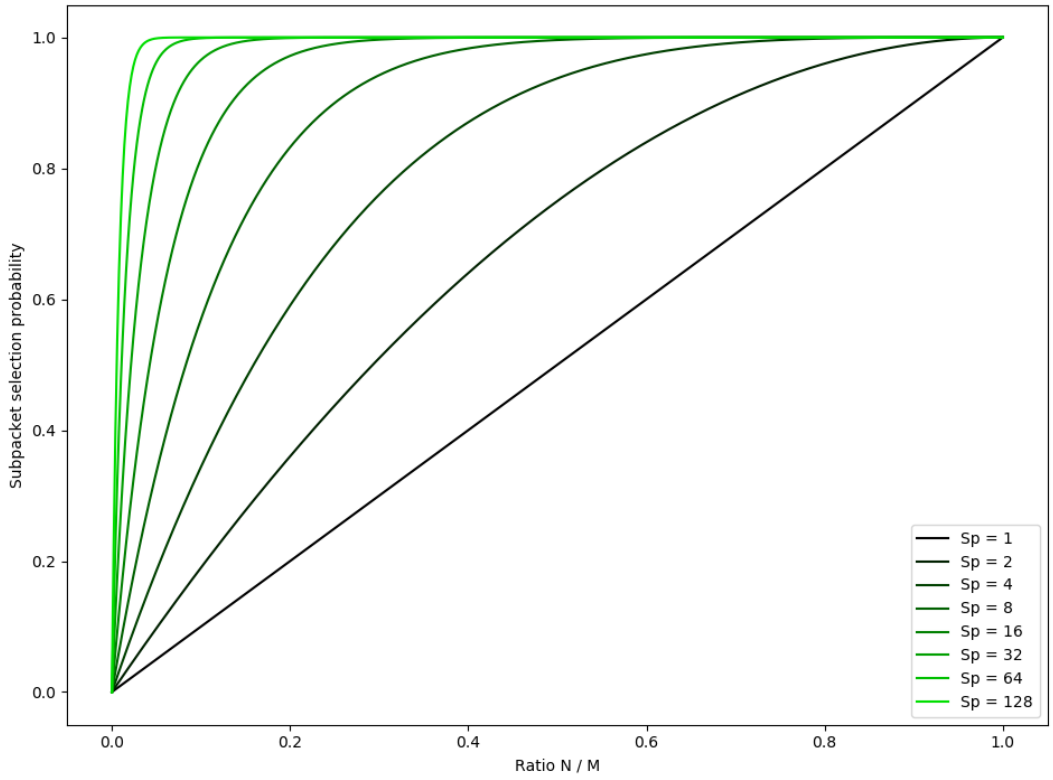


FIGURE 3.14: Probability that a subpacket of varying size S_p contains at least one spike, in function of the ration between N and M.

The output indexes histogram method being only dependent on the sparsity in input and weight spaces, this approach is best suited for neural networks with a very high number of neurons with high activations and weights sparsity. However the performances get rapidly worse as the sparsity is lowered in either space. While this implementation can be efficient simulating spiking neural networks for biological processes, where this method requirements are met for large scale

networks, machine-learning neural-layers remain relatively low-dimensional in comparison to these requirements. Examples of accurate deep learning model with a very high number of neurons per layer and brain-like sparsity remain to be designed in order to find a computational interest for such methods. Also, changing the synaptic connections in the context of learning may be computationally heavy, due to the list-format of synaptic storage in this method and the need for two distinct representations of the same list. For instance, if we consider four billion neurons in order to simulate the whole visual system, with one thousand inputs each, such network would take 16TB of memory, which is currently impossible to store on current GPU devices. However, this format allows a drastic reduction of synaptic storage with very sparse weights.

3.4.2 Discussion

We benchmarked three spike propagation algorithms on a minimalistic neural network to analyse the effect of a few parameters on processing times. Parameters studied were the input / output dimension, the input activations sparsity and the sparsity of the connections in the synaptic matrix. The main criticism which can be formulated about the simplifications we have done is the lack of complexity of our neural network compared to the other models used in neuroscience or deep learning. We argue first that such quantization can still be relevant considering the binary activation nature of spiking neural netwroks and recent advances in deep learning networks quantization (Courbariaux et al., 2016; Rastegari et al., 2016; Zhu et al., 2016; Deng et al., 2017). Also, in order to obtain the processing times function for a more complex network or on different hardware, one should apply some additional factors in the given equations. For instance, if we consider instead 32-bit floating point activations and synaptic weights, we should take into account that 32 times more memory loads and stores will occur, and AND-POPC operation would be replaced by 32 multiply-add operations. This would still give an estimate of the most useful algorithm for a given use-case.

One can argue with the relevance of our use of parametric optimization in order to obtain the memory access parameters values. Since it is obtained directly from the kernel processing time data, we are aware this introduces a bias in our study. For instance we found a column-read access latency of 1.7947 clocks (which should normally be an integer value) and a very low row-access latency compared to what is shown in GDDR5 constructors datasheets

(Micron Technology, 2014). While we tried to avoid the use of caching in our implementation, the CUDA compiler and the internal scheduler of GPU seemed to optimize such accesses anyway, leading to these low access latencies. However, the last parameter estimated, the kernel launch overhead of $14\mu s$, is perfectly in the ranges of what is seen for any CUDA implemented kernel launch. More transparency about NVidia technology would have allowed us to refine our approach. Anyway, the order of complexity obtained are consistent with what we have estimated.

While in this study we focused our attention on CUDA devices, we invite other researchers interested in estimating spike propagation times on various hardware to implement the proposed methods on many different parallel devices (such as FPGAs and AMD GPUs), preferably with more knowledge *a priori* on the memory latency parameters, in order to criticize and enrich the proposed computational models.

Given the growing interest in neuromorphic architectures, whether in the context of simulations of neural models or the deployment of neural networks learned with Deep Learning methods, it seems to us that our study is relevant in order to choose the best implementation for spike propagation according to the parameters of the network. Indeed, we hope that this benchmark will serve as a first guideline for people who wish to implement large-scale neural networks on massively parallel architectures.

Chapter 4

Unsupervised feature learning with Winner-Takes-All based STDP

Article presentation

In this chapter we explore how the STDP, the main learning mechanism in the brain, can be used to learn visual features while being compatible with GPUs architectures. We have seen in the state-of-the-art section 1 that biological learning rules are able to learn rapidly representations, which is a lacking feature in deep learning. This study follows research from Laurent Perrinet and Timothée Masquelier, who studied how visual representations may be learned in a biologically plausible manner.

Laurent Perrinet explored the learning of sparse codes with rank-order spiking neural networks (Perrinet et al., 2004; Perrinet, 2004). The approach proposed in these studies rely on the matching pursuit algorithm, which process input spikes sequentially to detect the maximum activity acrross all neurons. A network with lateral connections can learn V1 gabor-like features with this method (Perrinet, 2010). Masquelier and Thorpe (2007) proposed to learn unsupervisedly pattern in V2 specific to different classes with STDP. The network used in this study have two layers of simple cells, each with a complex cells layer on top of them. Having a Winner-Take-All mechanism based on the first neuron to spike, neurons converge rapidly toward discriminant parts of visual objects.

These studies, while showing how early visual features may emerge from experience, lacks of compatibility with parallel processing. Indeed, both matching pursuit and STDP need spike times to be iteratively processed one by one in

order to apply their respective competition mechanism. It is thus difficult to perform learning with a single feedforward propagation step as well as process multiple images in parallel.

We propose a method inspired from STDP that is able to learn rapidly visual features from batches of images which is compliant with GPUs architectures. We show that this method requires far less samples than deep learning methods to converge, and still reach state-of-the-art performance levels in classification tasks.

frontiers in Computational Neuroscience

METHODS
published: 05 April 2018
doi: 10.3389/fncom.2018.00024

Unsupervised Feature Learning With Winner-Takes-All Based STDP

Paul Ferré[1,2]*, *Franck Mamalet*[2] *and Simon J. Thorpe*[1]

[1] *Centre National de la Recherche Scientifique, UMR-5549, Toulouse, France,* [2] *Brainchip SAS, Balma, France*

We present a novel strategy for unsupervised feature learning in image applications inspired by the Spike-Timing-Dependent-Plasticity (STDP) biological learning rule. We show equivalence between rank order coding Leaky-Integrate-and-Fire neurons and ReLU artificial neurons when applied to non-temporal data. We apply this to images using rank-order coding, which allows us to perform a full network simulation with a single feed-forward pass using GPU hardware. Next we introduce a binary STDP learning rule compatible with training on batches of images. Two mechanisms to stabilize the training are also presented : a Winner-Takes-All (WTA) framework which selects the most relevant patches to learn from along the spatial dimensions, and a simple feature-wise normalization as homeostatic process. This learning process allows us to train multi-layer architectures of convolutional sparse features. We apply our method to extract features from the MNIST, ETH80, CIFAR-10, and STL-10 datasets and show that these features are relevant for classification. We finally compare these results with several other state of the art unsupervised learning methods.

Keywords: Spike-Timing-Dependent-Pasticity, neural network, unsupervised learning, winner-takes-all, vision

OPEN ACCESS

Edited by:
Guenther Palm,
Universität Ulm, Germany

Reviewed by:
Michael Beyeler,
University of Washington, United States
Stefan Duffner,
UMR5205 Laboratoire d'Informatique en Image et Systèmes d'Information (LIRIS), France

***Correspondence:**
Paul Ferré
paul.ferre@cnrs.fr

Received: *18 May 2017*
Accepted: *20 March 2018*
Published: *05 April 2018*

Citation:
Ferré P, Mamalet F and Thorpe SJ (2018) Unsupervised Feature Learning With Winner-Takes-All Based STDP. Front. Comput. Neurosci. 12:24. doi: 10.3389/fncom.2018.00024

1. INTRODUCTION

Unsupervised pre-training methods help to overcome difficulties encountered with current neural network based supervised algorithms. Such difficulties include : the requirement for a large amount of labeled data, vanishing gradients during back-propagation and the hyper-parameters tuning phase. Unsupervised feature learning may be used to provide initialized weights to the final supervised network, often more relevant than random ones (Bengio et al., 2007). Using pre-trained weights tends to speed up network convergence, and may also increase slightly the overall classification performance of the supervised network, especially when the amount of labeled examples is small (Rasmus et al., 2015).

Unsupervised learning methods have recently regained interest due to new methods such as Generative Adverserial Networks (Goodfellow et al., 2014; Salimans et al., 2016), Ladder networks (Rasmus et al., 2015), and Variational Autoencoders (Kingma and Welling, 2013). These methods reach state of the art performances, either using top layer features as inputs for a classifier or within a semi-supervised learning framework. As they rely on gradient descent methods to learn the representations for their respective tasks, computations are done with 32-bits floating point values. Even with dedicated hardware such as GPUs and the use of 16-bits half-floats type (Gupta et al., 2015), floating point arithmetic remains time and power consuming for large datasets. Several works are addressing this problem by reducing the resolution of weights, activations and gradients during inference and learning phases (Stromatias et al., 2015; Esser et al., 2016; Deng et al., 2017)

and have shown small to zero loss of accuracy with such supervised methods. Nevertheless, learning features both with unsupervised methods and lower precision remains a challenge.

On the other hand, Spiking Neural Networks (SNNs) propagate information between neurons using spikes, which can be encoded as binary values. Moreover, SNNs often use an unsupervised Hebbian learning scheme, Spike-Timing-Dependent-Plasticity (STDP), to capture representations from data. STDP uses differences of spikes times between pre and post-synaptic neurons to update the synaptic weights. This learning rule is able to capture repetitive patterns in the temporal input data (Masquelier and Thorpe, 2007). SNNs with STDP may only require fully feed-forward propagation to learn, making them good candidates to perform learning faster than backpropagation methods.

Our contribution is three-fold. First, we demonstrate that Leaky Integrate and Fire neurons act as artificial neurons (perceptrons) for temporally-static data such as images. This allows the model to infer temporal information while none were given as input. Secondly, we develop a winner-takes-all (WTA) framework which ensure a balanced competition between our excitatory neuron population. Third, we develop a computationally-efficient and nearly parameter-less STDP learning rule for temporally static-data with binary weight updates.

2. RELATED WORK

2.1. Spiking Neural Networks

2.1.1. Leaky-Integrate-and-Fire Model

Spiking neural networks are widely used in the neuroscience community to build biologically plausible models of neuron populations in the brain. These models have been designed to reproduce information propagation and temporal dynamics observable in cortical layers. As many models exists, from the most simple to the most realistic, we will focus on the Leaky-Integrate-and-Fire model (LIF), a simple and fast model of a spiking neuron.

LIF neurons are asynchronous units receiving input signals called spikes from pre-synaptic cells. Each spike x_i is modulated by the weight w_i of the corresponding synapse and added to the membrane potential u. In a synchronous formalism, at each time step, the update of the membrane potential at time t can be expressed as follow:

$$\mathcal{T}\frac{\delta u(t)}{\delta t} = -(u(t) - u_{res}) + \sum_{i=1}^{n} w_i x_{i,t} \tag{1}$$

Where $\mathcal{T}$ is the time constant of the neuron, n the number of afferent cells and u_{res} is the reset potential (which we also consider as the initial potential at $t_0 = 0$).

When u reaches a certain threshold T, the neuron emits a spike to its axons and resets its potential to its initial value u_{res}.

This type of network has proven to be energy-efficient Gamrat et al. (2015) on analog devices due to its asynchronous and sparse characteristics. Even on digital synchronous devices, spikes can be encoded as binary variables, therefore carrying maximum information over the minimum memory unit.

2.1.2. Rank Order Coding Network

A model which fits the criteria of processing speed and adaptation to images data is the rank order coding SNN (Thorpe et al., 2001). This type of network processes the information with single-step feed-forward information propagation by means of the spike latencies. One strong hypothesis for this type of network is the possibility to compute information with only one spike per neuron, which has been demonstrated in rapid visual categorization tasks (Thorpe et al., 1996). Implementations of such networks have proven to be efficient for simple categorization tasks like frontal-face detection on images (Van Rullen et al., 1998; Delorme and Thorpe, 2001).

The visual-detection software engine SpikeNet Thorpe et al. (2004) is based on rank order coding networks and is used in industrial applications including face processing for interior security, intrusion detection in airports and casino games monitoring. Also, it is able to learn new objects with a single image, encoding objects with only the first firing spikes.

The rank order model SpikeNet is based on a several layers architecture of LIF neurons, all sharing the time constant $\mathcal{T}$, the reset potential u_{res} and the spiking threshold T. During learning, only the first time of spike of each neuron is used to learn a new object. During inference, the network only needs to know if a neuron has spiked or not, hence allowing the use of a binary representation.

2.2. Learning With Spiking Neural Networks

2.2.1. Deep Neural Networks Conversion

The computational advantages of SNNs led some researchers to convert fully learned deep neural networks into SNNs (Diehl et al., 2015, 2016), in order to give SNNs the inference performance of back-propagation trained neural networks.

However, deep neural networks use the back-propagation algorithm to learn the parameters, which remains a computationally heavy algorithm, and requires enormous amounts of labeled data. Also, while some researches hypothesize that the brain could implement back-propagation (Bengio et al., 2015), the biological structures which could support such error transmission process remain to be discovered. Finally, unsupervised learning within DNNs remains a challenge, whereas the brain may learn most of its representations through unsupervised learning (Turk-Browne et al., 2009). Suffering from both its computational cost and its lack of biological plausibility, back-propagation may not be the best learning algorithm to take advantage of SNNs capabilities.

On the other hand, researches in neuroscience have developed models of unsupervised learning in the brain based on SNNs. One of the most popular model is the STDP.

2.2.2. Spike Timing Dependent Plasticity

Spike-Timing-Dependent-Plasticity is a biological learning rule which uses the spike timing of pre and post-synaptic neurons to update the values of the synapses. This learning rule is said to be Hebbian ("What fires together wires together").

Synaptic weights between two neurons updated as a function of the timing difference between a pair or a triplet of pre and post-synaptic spikes. Long-Term Potentiation (LTP) or a Long-Term Depression (LTD) are triggered depending on whether a presynaptic spike occurs before or after a post-synaptic spike, respectively.

Formulated two decades ago by Markram et al. (1997), STDP has gained interest in the neurocomputation community as it allows SNN to be used for unsupervised representation learning (Kempter et al., 2001; Rao and Sejnowski, 2001; Masquelier and Thorpe, 2007; Nessler et al., 2009). The features learnt in low-level layers have also been shown to be relevant for classification tasks combined with additional supervision processes in the top layers (Beyeler et al., 2013; Mozafari et al., 2017). As such STDP may be the main unsupervised learning mechanisms in biological neural networks, and shows nearly equivalent mathematical properties to machine learning approaches such as auto-encoders (Burbank, 2015) and non-negative matrix factorization (Carlson et al., 2013; Beyeler et al., in review).

We first consider the basic STDP pair-based rule from Kempter et al. (2001). Each time a post synaptic neuron spikes, one computes the timing difference $\Delta t = t_{pre} - t_{post}$ (relative to each presynaptic spike) and updates each synapse w as follows:

$$\Delta w = \begin{cases} A^+.e^{\frac{\Delta t}{\mathcal{T}_+}} \text{ if } \Delta t < 0 \\ A^-.e^{\frac{\Delta t}{\mathcal{T}_-}} \text{ otherwise} \end{cases} \quad (2)$$

where $A_+ > 0, A_- < 0$, and $\mathcal{T}_+, \mathcal{T}_- > 0$. The top and bottom terms in this equation are respectively the LTP and LTD terms.

This update rule can be made highly computationally efficient by removing the exponential terms $e^{\frac{\Delta t}{\mathcal{T}}}$, resulting in a simple linear time-dependent update rule.

Parameters A_+ and A_- must be tuned on order to regularize weight updates during the learning process. However in practice, tuning these parameters is a tedious task. In order to avoid weight divergences, networks trained with STDP learning rule should also implement stability processes such as refractory periods, homoeostasis with weight normalization or inhibition. Weight regularization may also be implemented directly by reformulating the learning rule equations. For instance in Masquelier and Thorpe (2007), the exponential term in Equation (2) is replaced by a process which guaranties that the weights remain in the range [0...1] :

$$\Delta w = \begin{cases} A_+.w.(1-w) \text{ if } \Delta t < 0 \\ A_-.w.(1-w) \text{ otherwise} \end{cases} \quad (3)$$

Note that in Equation (3), the amplitude of the update is independent from the absolute time difference between pre and post-synaptic spikes, which only works if pairs of spikes belongs to the same finite time window. In Masquelier and Thorpe (2007) this is guaranteed by the whole propagation schemes, which is applied on image data and rely on a single feedforward propagation step taking into account only one spike per neuron. Thus the maximum time difference between pre and post-synaptic spikes is bounded in this case.

2.3. Regulation Mechanisms in Neural Networks

2.3.1. WTA as Sparsity Constrain in Deep Neural Networks

Winner-takes-all (WTA) mechanisms are an interesting property of biological neural networks which allow a fast analysis of objects in exploration tasks. Following de Almeida et al. (2009), gamma inhibitory oscillations perform a WTA mechanism independent from the absolute activation level. They may select the principle neurons firing during a stimulation, thus allowing, e.g., the tuning of narrow orientation filters in V1.

WTA has been used in deep neural networks in Makhzani and Frey (2015) as a sparsity constraint in autoencoders. Instead of using noise or specific loss functions in order to impose activity sparsity in autoencoder methods, the authors propose an activity-driven regularization technique based on a WTA operator, as defined by Equation (4).

$$WTA(X,d) = \begin{cases} X_j \text{ if } |X_j| = \underset{k \in d}{max}(|X_k|) \\ 0 \text{ otherwise} \end{cases} \quad (4)$$

where X is a multidimensional matrix and d is a set of given dimensions of X.

After definition of a convolutional architecture, each layer is trained in a greedy layer-wise manner with representation from the previous layer as input. To train a convolutional layer, a WTA layer and a deconvolution layer are placed on top of it. The WTA layer applies the WTA operator on the spatial dimensions of the convolutional output batch and retains only the n_p% first activities of each neuron. This way for a given layer with N representations map per batch and C output channels, only $N.n_p.C$ activities are kept at their initial values, all the others activation values being zeroed. Then the deconvolutional layer attempts to reconstruct the input batch.

While this method demonstrates the potential usefulness of WTA mechanisms in neural networks, it still relies on computationally heavy backpropagation methods to update the weights of the network.

2.3.2. Homosynaptic and Heterosynaptic Homeostasis

In their original formulation, Hebbian-type learning rule (STDP, Oja rule , BCM rule) does not have any regulation process. The absence of regulation in synaptic weights may impact negatively the way a network learns. Hebbian learning allows the synaptic weights to grow indefinitely, which can lead to abnormally high spiking activity and neurons to always win the competitions induced by inhibitory circuits.

To avoid such issues, two types of homeostasis have been formulated.

Homosynaptic homeostasis acts on a single synapse and is depends on its respective inputs and outputs activity only. This homeostatic process can be modeled with a self-regulatory term in the Hebbian rule as in Masquelier and Thorpe (2007) or as a synaptic scaling rule depending on the activity driven by the synapse as in Carlson et al. (2013).

Heterosynaptic homeostasis is a convenient way to regulate the synaptic strength of a network. The model of such homeostasis takes into account all the synapses connected to a given neuron, all the synapses in a layer (like the L2 loss weight decay in deep learning) or at the network scale. Biological plausibility of such process is still discussed. Nevertheless, some evidences of heterosynaptic homeostasis have been observed in the brain to compensate runaway dynamics of synaptic strength introduced by Hebbian learning (Royer and Paré, 2003; Chistiakova et al., 2014). It then plays an important role in the regulation of spiking activity in the brain and is complementary to homosynaptic plasticity.

2.4. Neural Networks and Image Processing

Image processing with neural networks is performed with multiple layers of spatial operations (like convolutions, pooling, and non-linearities), giving the name Deep Convolutional Neural Networks to these methods. Their layer architecture is directly inspired from the biological processes of the visual cortex, in particular from the well known HMAX model (Riesenhuber and Poggio, 1999), except that the layers' weights are learnt with back-propagation. Deep CNN models use a single-step forward propagation to perform a given task. Even if convolutions on large maps may be computationally heavy, all the computations are done through only one pass in each layer. One remaining advantage of CNNs is their ability to learn from raw data, such as pixels for images or waveforms for audio.

On the other hand, since SNNs use spikes to transmit information to the upper layers, they need to perform neuron potential updates at each time step. Hence, applying such networks with a convolutional architecture requires heavy computations once for each time step. However, spikes and synaptic weights may be set to a very low bit-resolution (down to 1 bit) to reduce this computational cost Thorpe et al. (2004). Also, STDP is known to learn new representations with a few iterations Masquelier et al. (2009), theoretically reducing the number of epochs required to converge.

3. CONTRIBUTION

Our goal here is to apply STDP in a single-step feed-forward formalism directly from raw data, which should be beneficial in the cases where training times and data labeling are issues. Thus we may select a neural model which combines the advantages of each formalism in order to reduce the computational cost during both training and inference.

3.1. Feedforward Network Architecture

3.1.1. Neural Dynamics

Here, we will consider the neural dynamics of a spiking LIF network in presence of image data. Neural updates in the temporal domain in such neural architecture are as defined by Equation (1).

Since a single image is a static snapshot of visual information, all the $x_{i,t}$ are considered constant over time. Hence $\sum_{i=1}^{n} w_i.x_{i,t}$ is also constant over time under the assumption of static synaptic weights during the processing of the current image.

Let us define $v_{in} = \sum_{i=1}^{n} w_i.x_{i,t}, \forall t$ the total input signal to the neuron. Let us also determine $u(t_0 = 0) = u_{res}$ as an initial condition. As v_{in} is constant over time, we can solve the differential equation of the LIF neuron, which gives:

$$\begin{aligned} &\mathcal{T}\frac{\delta u(t)}{\delta t} = -(u(t) - u_{res}) + v_{in} \\ &\Rightarrow \quad u(t) = -v_{in}.e^{\frac{-t}{\mathcal{T}}} + u_{res} + v_{in} \qquad \forall t > 0 \end{aligned} \tag{5}$$

The precise first spike-time of a neuron given its spiking threshold T is given by :

$$t_s = -\mathcal{T}.log(1 + \frac{u_{res} - T}{v_{in}}) \tag{6}$$

Since Equation (6) decreases monotonically wrt. v_{in}, we can recover the intensity-latency equivalence. The relative order of spike-times is also known since $v_{in,1} > v_{in,2} \rightarrow t_{s,1} < t_{s,2}$.

3.1.2. Equivalence With Artificial Neuron With ReLU Activation

Thus from Equation (6), for each neuron we can determine the existence of a first spike, along with its precise timing. Hence, since we are only concerned with the relative times of first spikes across neurons, one can replace the computation at each time-step by a single-step forward propagation given the input intensity of each neuron.

The single-step forward propagation correspond to LIF integration when $t \rightarrow \infty$. As we are first looking for the existence of any t_s such that $u(t_s) > T$:

$$\begin{aligned} \lim_{t\to\infty} u(t) - T &= \lim_{t\to\infty} -v_{in}.e^{\frac{-t_s}{\mathcal{T}}} + u_{res} + v_{in} - T \\ &= u_{res} + v_{in} - T \end{aligned} \tag{7}$$

Having $v_{in} = \sum_{i=1}^{n} w_i.x_i$ and $b = u_{res} - T$,

$$\lim_{t\to\infty} u(t) - T = b + \sum_{i=1}^{n} w_i.x_i \tag{8}$$

which is the basic expression of the weighted sum of a perceptron with bias.Also, t_s exists if and only if $b + \sum_{i=1}^{n} w_i.x_i > 0$, which shows the equivalence between LIF neurons with constant input at infinity and the artificial neuron with rectifier activation function (ReLU).

This demonstration can be generalized to local receptive fields with weight sharing, and thus we propose to replace the time-step computation of LIF neurons, by common GPU optimized routines of deep learning such as 2D convolutions and ReLU non-linearity. This allows us to obtain in a single-step all the first times of spikes -inversely ordered by their activation level- and nullified if no spike would be emitted in an infinite time. Moreover, these different operations are compatible with mini-batch learning. Hence, our model is also capable of processing

several images in parallel, which is an uncommon feature in STDP-based networks.

3.1.3. Winner-Takes-All Mechanisms

Following the biological evidence of the existence of WTA mechanisms in visual search tasks (de Almeida et al., 2009) and the code sparsity learned with such processes (Makhzani and Frey, 2015), we may take advantage of WTA to match the most repetitive patterns in a given set of images. Also, having to learn only these selected regions should drastically decrease the number of computations required for the learning phase (compared to dense approaches in deep learning and SNN simulations). Inspired by this biological mechanism, we propose to use three WTA steps as sparsifying layers in our convolutional SNN architecture.

The first WTA step is performed on feature neighborhood with a max-pooling layer on the convolution matrix with kernel size $k_{pool} >= k_{conv}$ and stride $s_{pool} = k_{conv}$. This acts as a lateral inhibition, avoiding the selection of two spikes from different kernels in the same region.

Next we perform a WTA step with the WTA operation (Equation 4) on the channel axis for each image (keeping at each pooled pixel, the neuron that spikes first). This forces each kernel to learn from different input patches.

The third WTA step is performed with WTA operation on spatial axes as in Makhzani and Frey (2015). This forces the neuron to learn from the most correlated patch value in the input image.

The WTA operation (Equation 4) is not to be confused with the Maxout operation from Goodfellow et al. (2013) and the max pooling operation, since these latter squeeze the dimensions on which they are applied, while the WTA operation preserves them.

Then we extract the indexes of the selected outputs along with their sign and their corresponding input patch. Extracted input patches are organized in k subsets, each subset corresponding to one output channel. These matrices will be refered to as follow :

- Y_k : matrices of selected outputs, of dimension (m_k, c_{out})
- X_k : matrices of selected patches, of dimension $(m_k, c_{in} \times h_{in} \times w_{in})$
- W : matrices of filters, of dimension $(c_{in} \times h_{in} \times w_{in}, c_{out})$

with m_k the number of selected indexes and patches for neuron $k \in [1...c_{out}]$, c_out the number of channels (or neurons) of the output layer, and c_in, h_in, w_in are the receptive field size (resp. channel, height and width). Note that at most one output is selected per channel and per image, $m_k \leq N$.

The WTA in our model has two main advantages. First, it allows the network to learn faster on only a few regions of the input image. Second, classical learning frameworks use the mean of weights gradient matrix to update the synaptic parameters. By limiting the influence of averaging on the gradient matrix, synaptic weights are updated according to the most extreme values of the input, which allow the network to learn sparse features.

Note that the network is able to propagate relative temporal information through multiple layer, even though presented inputs lack this type of data. It is also able to extract regions which are relevant to learn in terms of information maximization. The full processing chain for propagation and WTA is shown in **Figure 1**.

3.2. Binary Hebbian Learning

3.2.1. Simplifying the STDP Rule

Taking inspiration from the STDP learning rule, we propose a Hebbian correlation rule which follows the relative activations of input and output vectors.

Considering the input patch value $x_{n,i} \in X_n, n \in [1...m_k], i \in [1...c_{in} \times h_{in} \times w_{in}]$, the corresponding weight value $w_{k,i}$, the selected output value $y_k \in Y_k$ and a heuristically defined threshold T_l, the learning rule is described in Equation (9).

$$\Delta w_{k,i} = \begin{cases} sign(x_{n,i}).sign(y_k) \text{ if } |x_{n,i}| > T_l \\ -sign(w_{k,i}) \text{ otherwise} \end{cases} \tag{9}$$

The learning rule is effectively Hebbian as shown in the next paragraph and can be implemented with lightweight operations such as thresholding and bit-wise arithmetic.

Also, considering our starting hypotheses, where we limit to one the number of spikes per neuron during a full propagation phase for each image, it is guaranteed that, for any pair of pre and post-synaptic neuron, the choice of LTP or LTD exist and is unique for each image presentation. These hypotheses are similar to the ones in Masquelier and Thorpe (2007), where these conditions simulates a single wave of spikes within a range of 30 ms.

3.2.2. Equivalence to Hebbian Learning in Spiking Networks

In this section we show the Hebbian behavior of this learning rule. For this, we first focus on the "all positive case" $(x, y, w \in R+)$ and will explain in the next section the extension to symmetrical neurons.

In the case of "all positive," the Equation (9) can be rewritten as Equation (10).

$$\Delta w_{k,i} = \begin{cases} 1 \text{ if } x_{k,i} > u(t_{post}) \\ -1 \text{ otherwise} \end{cases} \tag{10}$$

This rule tends to increase the weights when the input activity is greater than a threshold (here the post-synaptic neuron firing threshold), and decreases it otherwise.

Equation (10) is equivalent to the pair-based STDP rule presented in Equation (2) removing the exponential term and using $A_+ = 1$ and $A_- = -1$.

3.2.3. Extension to Symmetric Neurons

We have demonstrated that the proposed learning rule is effectively Hebbian in the case where $x, w, y \in \mathbb{R}_+$. Our learning rule also takes into account negative values of x, w, y. In biological networks models, negative values do not seem to make much

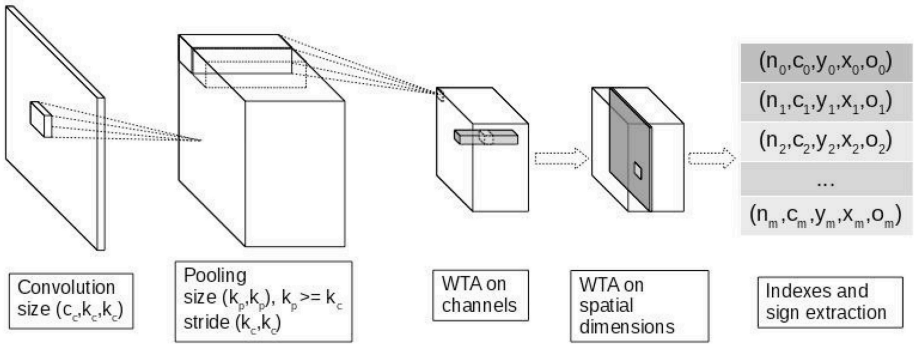


FIGURE 1 | Processing chain for the region WTA.

TABLE 1 | Weight update given x, y, and w following the proposed learning rule (Equation 9).

	$x < -T$	$-T < x < T$	$x > T$
$y > 0$	-1	$-sign(w)$	$+1$
$y < 0$	$+1$	$-sign(w)$	-1

sense since firing rates and synaptic conductance are expressed in units defined only in $\mathbb{R}_+$.

Nevertheless, negative values are used in many spiking networks models in the very first layer of visual features. For instance, ON-centered-OFF-surround and OFF-centered-ON-surround filters (also known as *Mexican hat* filters) are often used to pre-process an image in order to simulate retinal cells extracting gradients. These two filters are symmetric with respect to the origin. Hence a common computational optimization is to apply only one of the two filters over the image, separating negative and positive resulting values as OFF and ON activities, respectively.

We extend this computational trick to neurons in any neural layer under the hypothesis that negative values for x, w, y corresponds to activities and weights of synaptically symmetric neurons. For a neuron with constant input activity X and synaptic weights W of size n, we can express its output activity $y = \sum_{i=1}^{n} X_i \times W_i$. If $y < 0$, we can convert it to a positive value using the synaptically opposite weights $\sum_{i=1}^{n} X_i \times -W_i = -y$.

Under the hypothesis of the existence of a pair-wise competition between neurons with symmetric weights (for instance with inhibition), this computational trick remains biologically plausible.

Considering now the proposed learning rule, the weights update given x, y, and w is shown in **Table 1**. In this table, the first spikes ($|x| > T$) will induce an update of the weight to increase the $|y|$ ($\Delta w = sign(y).sign(x)$). Meanwhile, the weights corresponding to the last spike will be reduced ($\Delta w = -sign(w)$).

With this framework the choice of the parameter T_l is critical. Thanks to the WTA mechanism developed, the selection of a neuron for learning is performed disregarding its firing threshold T, set to zero in practice. Hence contrary to Masquelier and Thorpe (2007), we cannot rely on the precise firing threshold of the neuron. In order to approximate this threshold, we developed two strategies described in the next paragraphs. These strategies are made adaptative such that the learning rule can be invariant to contrast variation. Also the adaptative behavior of this threshold avoids to tune an additional parameter in the model.

3.2.4. Hard Percentile Threshold

The first strategy applied follows the STDP learning rule, which fixes a time constant for LTP and LTD. In our framework this is implemented as a percentile of the input activity to map their influence in the spike. For each input vector $x_n \in X_k \forall k$, we compute the patch threshold T_l as the minimum value in the local $p_{n\%}$ percentile. $p_{n\%}$ is manually set and global for all the patches.

$$\Delta w_{k,i} = \begin{cases} -sign(w_{k,i}) \text{ if } |x_{n,i}| \leq p_{n\%} \\ sign(x_{n,i}).sign(y_k) \text{ otherwise} \end{cases} \tag{11}$$

However, we have seen experimentally that the threshold tuning may be cumbersome. As it regulates the sparsity of the synaptic weight matrix, fixing the sparsity manually may lead to unsatisfying results. Also, getting the percentiles uses the index-sorting operation which is time consuming.

3.2.5. Average Correlation Threshold

We propose a second strategy which relies on the computation of an adaptative threshold between LTP and LTD. For each input vector $x_n \in X_k \forall k$ we compute the sign correlated input activation as $\hat{x_{n,i}} = x_{n,i}.sign(w_k).sign(y_k)$. Next we compute the threshold T_l as the mean of $\hat{x_n}$. Then we apply the learning rule in Equation (9).

With this strategy, the learning rule is also equivalent to Equation (12), which is straightforward to implement since it avoids conditional branching.

$$\Delta w_{k,i} = sign(x_{n,i}.sign(y_k).sign(w_{k,i}) - T_l).sign(w_{k,i}) \tag{12}$$

Using the mean sign corrected input activation as a threshold, the model is able to be invariant to local contrasts. It also requires the

calculation of the mean and a thresholding, two operations that are much faster than sorting. Finally, the adaptative behavior of such a threshold automate the sparsity of synaptic weights.

3.2.6. Computing Updates From a Batch of Images

Since our method allows the propagation of several images at the same time through mini-batch, we can also adapt our learning rule when batches of images are presented. Since biological visual systems never deal with batches of dozen images at once, the following proposal is a computational trick to accelerate the learning times, not a model of any existing biological feature.

When all the update vectors have been computed, the weight update vector for the current batch is obtained through the binarization of the sum of all the update vector for the corresponding kernel. We finally modulate the update vector with a learning rate λ.

$$U_{n,i} = \sum_{k=1}^{m_k} \Delta w_{k,i} \tag{13}$$

$$\Delta W_{k,i} = \begin{cases} -1 \text{ if } U_{n,i} \leq 0 \\ 1 \text{ otherwise} \end{cases} \tag{14}$$

$$W_{k,i} = W_{k,i} + \lambda . \Delta W_{k,i} \tag{15}$$

3.2.7. Weight Normalization Through Simple Statistics

Since each update step adds $+\lambda$ or $-\lambda$ to the weights, a regularization mechanism is required to avoid the weights growing indefinitely. Also we want to maintain a fair competition between neurons of the same layer, thus the total energy of the weights should be the same for all the neurons.

We propose a simple model of heterosynaptic homeostasis in order to regulate the weights of each neuron.We chose to normalize the weights of each neuron k by mean centering and standardization by variance. Hence, after each update phase, the normalization is done as follows :

$$W_k = \frac{W_k - \mu(W_k)}{\sigma^2(W_k)} \tag{16}$$

This way, even neurons which did not learn a lot during the previous epochs can win a competition against the others. In practice, we set λ in an order of magnitude of 10^{-1} and halved it after each epoch. Given the order of magnitude of λ and the unit variance of W_k, we know that ninety-five percent of the weights belongs to the interval $[-1.5...1.5]$. In fact, only a few batches of images are necessary to modify the influence of a given afferent. Two neurons responding to a similar pattern can thus diverge and specialize on different patterns in less than a dozen training batches.

As a detail, if the WTA region selected is small, some neurons may learn parts of patterns already learned by an other one. Since $\sigma^2(W_k) = 1$ and most of the weights are equal to zero, the values of the remaining weights would grow very large. This can end up in multiple neurons learning almost identical patterns. We have observed that clipping weights after normalization between the range $[-2...2]$ prevents this situation.

3.3. Multi-layer Architectures With Binary STDP

This proposed approach is able to learn a multi-layer convolutional architecture as defined by the user. It does not require a greedy layer-wise training, all the convolutional layers can be trained in parallel. We can optionally apply a non-linearity, a downsampling operation or a normalization after each convolution layer.

Once all the features layers have learned, the whole features architecture can process images as a classical convolutional neural network in order to obtain the new representations.

4. EXPERIMENTS AND RESULTS

4.1. Method

The proposed method learns, unsupervised, convolutional features from image data. In order to validate our approach, we evaluated the learnt features on four different classification datasets : MNIST, ETH80, CIFAR10, and STL10. Architectures and hyper-parameters were tuned separately for each dataset, details being given in the relevant sections.

The overall evaluation method remains the same for each dataset. The proposed framework will be used to learn one or several convolutional layer with the simplified STDP. In order to show the faster convergence of features with our method, we will only train these layer with a subset of the full training dataset with very few epochs.

Once the features are learnt, we show qualitatively the learnt features for each dataset. To quantitatively demonstrate their relevance, we use the extracted features as input to a supervised classifier. Although as state of the art classification are deep learning systems, we use a simple Multi-Layer Perceptron (MLP) with zero, one, or two hidden layers (depending on the dataset) taking as inputs the learnt features with the proposed solution.

For all the experiments, we started with a lightweight network architecture (the simplest available in the literature if available), and incrementally added complexity until further additions stopped improving performance. The classifier on top of the network starts as linear dense layer with as many neurons as the number of classes, and is complexified with intermediate layers as the architectural-tuning goes on.

We compare our results with other state of the art unsupervised feature learning methods specific for each dataset.

4.2. MNIST

The MNIST dataset contains 60,000 training images and 10,000 testing images of size 28×28 containing handwritten digits from 0 to 9. MNIST digits are written in white on a black background, hence pixel values are distributed across two modes. Considering the data distribution and the limited number of classes, MNIST may be considered as an easy classification task for current state-of-the-art methods. As a matter of fact, neural based methods do not need deep architectures in order to perform well on this dataset. Light-weight architectures can be defined in order to explore issues with the developed method. Once the method has satisfying results on MNIST, more complex datasets may be tackled.

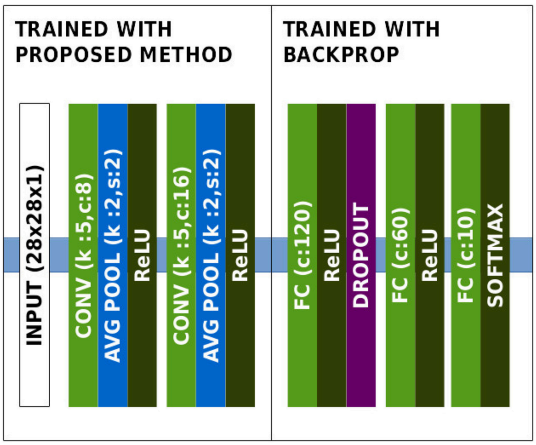


FIGURE 2 | Architecture of the network in the MNIST experiment.

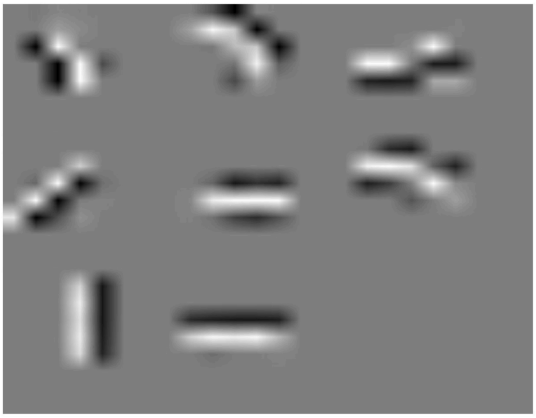


FIGURE 3 | Eight 5 × 5 features learned from MNIST dataset on raw images.

To perform classification on this dataset, we defined a lightweight convolutional architecture of features close to LeNet LeCun et al. (1998), presented in **Figure 2**. Since achieving high classification accuracy on MNIST is easy with a high number of neurons per layer, the number of neurons per layer was kept as low as possible in order to actually verify the relevance of the features.

Unsupervised learning was performed over only 5,000 random images from the dataset for 5 epochs, which only represents 25,000 image presentations. A visualization of the learnt features is shown in **Figure 3**.

Once the features were learnt, we used a two-hidden layers MLP to perform classification over the whole transformed training set. The learnt features and classifier were then run on all the testing set images in order to get the test error rate.

Classification performances are reported in **Table 2**. While the best methods in the state-of-the-art reach up to 99.77% accuracy, we did not report these results since these approaches use supervised learning with data augmentation, which is outwith the scope of this paper. All the reported results were obtained without data augmentation and using unsupervised feature learning.

TABLE 2 | MNIST accuracy.

Method	Accuracy (%)
SDNN (Kheradpisheh et al., 2016)	98.40
Two layer SNN (Diehl and Cook, 2015)	95.00
PCA-Net (Chan et al., 2014)	98.94
Our method	98.49

Our approach performs as well as SDNN since they are structurally close, reaching state-of-the-art performance without fine-tuning and data-augmentation. While PCA-Net has better performance, learning was done on twice the number of samples we used. Doubling the number of samples to match the same number used for PCA-Net (10,000) did not improve the performance of our method.

4.3. ETH80

The ETH80 (Leibe and Schiele, 2003) contains 3,280 color images of eight different object categories (apple, car, cow, cup,dog, horse, pear, tomato). Each category contains 10 different object instances taken from 41 points of view. This dataset is interesting since the number of available images is limited and contains a lot of variability in 3D rotations. It allows us to evaluate the generalization potential of the features and their robustness to changes in viewpoint.

As the number of samples is restrained here, we performed both unsupervised and supervised learning on half the dataset (1,640 images chosen randomly). The other half was used as the test set.

We compare our approach to the classical HMAX model and to Kheradpisheh et al. (2016). The architectures for unsupervised and supervised part are shown in **Figure 4**. Learning visual features becomes more and more difficult with the proposed method as we add convolutional layers on top of the network. Since ETH80 images are large (96 × 96), we apply pooling with a stride of 4 in order to quickly reduce the dimensions over the hierarchy.

Results are reported in **Table 3**. While our approach does not reach the same performance as Kheradpisheh et al. (2016), it is able to learn features relevant for a classification task with multiple points of view of different objects.

4.4. CIFAR-10

The CIFAR-10 dataset (Krizhevsky, 2009) is a dataset for classification of natural images from 10 classes (airplane, automobile, bird, cat,deer, dog, frog, horse, ship, and truck). The dataset is split into three with 60,000 training, 10,000 validation, and 10,000 testing images. Images are a subset of the 80 million tiny images dataset (Torralba et al., 2008). All the images are 32 × 32 pixels size with three color channels (RGB).

This dataset is quite challenging, since it contains many variations of objects with natural backgrounds, in low resolution.

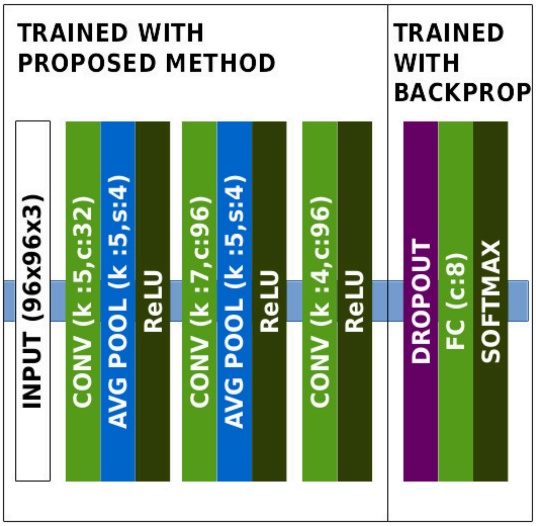


FIGURE 4 | Architecture of the network in the ETH80 experiment.

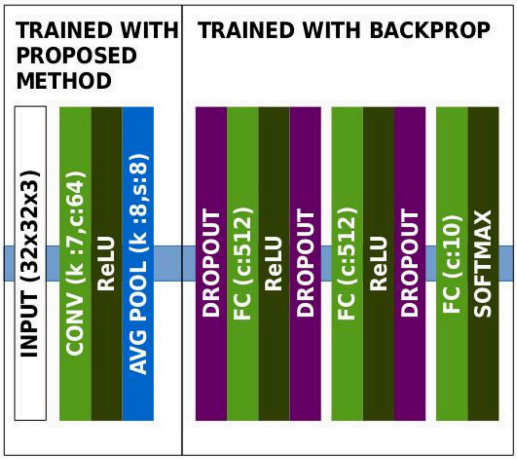


FIGURE 5 | Architecture of the network in the CIFAR-10 experiment.

TABLE 3 | ETH80 results.

Method	Accuracy (%)
HMAX (Riesenhuber and Poggio, 1999)	69.0
SDNN (Kheradpisheh et al., 2016)	82.8
Our method	75.2

Hence in order to tackle this dataset, algorithms must be able to find relevant information in noisy data.

The architecture used for this dataset is given in **Figure 5**. Learnt features are shown in **Figure 6A**. We observe that the features are similar to oriented-gabor features, which is consistent with the results of other unsupervised methods such as *k*-means and RBM. Also the weights distribution displayed in **Figure 6B** contains a majority of values close to zero, showing the sparsity of the features. Performances obtained on CIFAR-10, along with other methods evaluation, are shown in **Table 4**.

As a performance baseline, we also trained the MLP with the same architecture but keeping the convolutional layer's weights randomly initialized and frozen. The increase of 17% of classification rate proves the usefulness of the features learnt with our method in the classification process.

Only a few works related to SNNs have been benchmarked on CIFAR-10. Cao et al. (2015) and Hunsberger and Eliasmith (2015) rely on convolutional to SNN conversion to perform supervised learning on the dataset. Panda and Roy (2016) built a convolutional feature hierarchy on the principle of auto-encoders with SNNs, and classified the top level representations with an MLP.

Also, some works unrelated to SNNs are worth comparing here. Coates et al. (2011) benchmarked four unsupervised feature learning methods (k-means, triangle k-means, RBM, and sparse auto-encoders) with only one layer. Results from the PCA-Net approach are also included.

Our approach reached good performance given the lightweight architectures and the limited number of samples. It outperforms the CNN with 64 random filters, confirming the relevance of the learnt features for classification, and also the Triangle K-means approach with 100 features. Empirically however, training with more samples without increasing the number of features does not improve the performance.

Also, due to the low resolution of CIFAR-10 images, we tried to add a second convolutional layer. The learnt filters in this new layer were very redundant and led to the same performance observed with only one layer. Further investigations might explore ways to force layers above the first to learn more sparse features.

4.5. STL-10

STL-10 is a dataset dedicated to unsupervised feature learning. Images were taken from the ImageNet dataset. The training set contains 5,000 images labeled over the same ten classes as CIFAR-10. An unlabeled training set of 100,000 images is also provided. Unlabeled images may contain objects from other classes of ImageNet (like bear, monkeys, trains...). The testing set contains 8,000 images (800 per class). All images are in RGB format with a resolution of 96 × 96.

We applied the same architecture as for the CIFAR-10 dataset, except the average pooling layer was done over 24 × 24 sized windows (in order to have the same 4 × 4 output dimension). As before, we limited the number of samples during the unsupervised learning step to 5,000.

While some works related to SNNs or STDP have been benchmarked on CIFAR-10, we were not able to find any using the STL-10 dataset. Hence our approach may be the first biologically inspired method trying to tackle this dataset.

Our approach reaches 60.1% accuracy on STL-10, which is above the lower-bound performance on this dataset. Performances obtained by other unsupervised methods range between 58 and 74%.

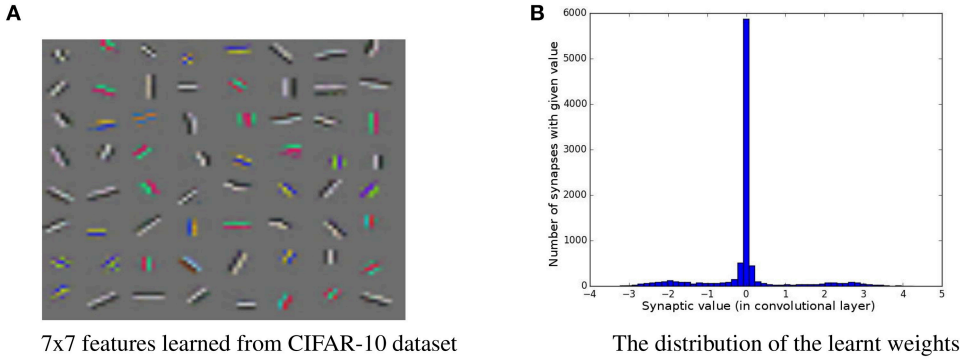


FIGURE 6 | (A) Sixty-four filters of size 7 × 7 learned with our method on the CIFAR-10 dataset. **(B)** The weights distribution of the network's first layer trained on CIFAR-10.

TABLE 4 | CIFAR-10 results.

Method	Unsupervised	Training samples	Accuracy (%)
Triangle k-means (1,600 features) (Coates et al., 2011)	Yes	50,000	79.6
Triangle k-means (100 features) (Coates et al., 2011)	Yes	50,000	55.5
PCA-Net (Chan et al., 2014)	Yes	50,000	78.67
LIF CNN (Hunsberger and Eliasmith, 2015)	No	50,000	82.95
Regenerative Learning (Panda and Roy, 2016)	Yes	20,000	70.6
Our method (64 features)	Yes	5,000	71.2
CNN random frozen filters	No	50,000	55.3

5. DISCUSSION

The proposed approach is able to train lightweight convolutional architectures based on LIF neurons which can be used as a feature extractor prior to a supervised classification method. These networks achieve average levels of performance on four image classification datasets. While the performances are not as impressive as the ones obtained with fully supervised learning methods, where features are learnt specifically for the classification task, interesting characteristics emerge from this model.

By showing the equivalence between rank-order LIF neurons and perceptrons with ReLU activation, we were able to borrow computationally efficient concepts from both neuroscience and machine learning literature while remaining biologically plausible enough to allow the conversion of network trained this way to be converted into SNN.

Binary STDP along with WTA and synaptic normalization reduces drastically the process of parameters tuning compared to other STDP approaches. LIF neurons require the tuning of their respective time constant. STDP also requires four parameters to be tuned : the time constants $\mathcal{T}_+$ and $\mathcal{T}_-$ as well as the LTP and LTD factors A_+ and A_- for each layer. Our model of binary STDP on the other hand only needs to set its learning rate λ, set globally for the whole architecture.

Another advantage over other STDP approaches is the ability to train the network with multiple images in parallel. While this ability is biologically implausible, it can become handy in order to accelerate the training phase thanks to the intrinsic parallel optimization provided by GPU. Also, the equivalence between LIF neurons and perceptrons with ReLU activation in presence of images allows us to perform the full propagation phase of a SNN in one shot, and to apply our STDP rule without the need of interpolation precise timing information from the image. Other approaches using SNNs with STDP requires the interpolation of temporal information from the image (Masquelier and Thorpe, 2007; Kheradpisheh et al., 2016), with gabor filters for instance, in order to generate spike trains. This way, STDP can be applied to learn the correlations between spike timings.

From a deep learning point of view, the main interest of our model resides in the proposal of a backpropagation-free training procedure for the first layers. As the backward pass in deep neural networks implies computationally heavy deconvolutions to compute the gradients of the parameters, any prior on visual modelization which can avoid a backpropagation over the whole network may help to reduce the computational overhead of this step. The LIF-ReLU equivalence demonstrated allows a convolutional network to take advantage of the inherent characteristic of STDP to quickly find repeating pattern in an input signal (Masquelier and Thorpe, 2007; Masquelier et al., 2009; Nessler et al., 2009).

With the WTA scheme proposed, we made the assumption that relevant visual information resides in the most contrasted patches. It also imposes the neurons to learn a sparse code with the combination of neighburhood and channel-wise inhibition. Such hard-coded WTA led to first layers features very similar to the gabor-like receptive-fields of LGN and V1. Quantitatively, the performances obtained on classification tasks allows us to conclude on the relevance of this learning process on such task. However it is still far from optimality considering the supervised learning methods (Graham, 2014; Hunsberger and Eliasmith, 2015) and human-level performances. The main drawback of our method is the difficulty to train more than one or two convolutional layers with. Since spatial inhibitions are critical in our WTA scheme to achieve feature sparseness, we suspect that

the input width and height of one layer must be large enough to make the competition between neurons effective. Other competition schemes less dependent on the spatial dimension have to be explored in order to train deeper architectures with the proposed framework.

Also our binary variant of STDP rule shows the ability to train neurons with very low precision updates. Gradients used to be coded on floating-point variables ranging from 32 bits as these encoding schemes had the better trade-off between numerical precision and efficiency on CPU and GPU hardware. Gupta et al. (2015) showed the possibility to perform gradient descent with only 16-bits floating-point resolution, a feature implemented since then in NVidia Pascal and AMD RX Vega GPUs. Studies on gradient quantization (Zhou et al., 2016; Deng et al., 2017) showed promising results reducing the precision down to 2 bits without penalizing significantly the performances. The main advantage of such reduction in resolution is two-fold : the lowest the resolution, the fastest the computations (under the condition hardware has sufficient dedicated compute units) and the fastest the memory transfers. Seide et al. (2014) accelerated learning speed by a factor 50 quantizing the weight updates gradients on 1 bit, enabling a very fast transfer between the 8 GPU of the considered cluster. The binary STDP learning rule proposed here may fit this goal. Further quantization on activations and weights (even if the distributions obtained on MNIST and CIFAR-10 seem to converge to three modes) are to be studied in such framework in order to bring massive acceleration thanks to this biologically inspired method.

In order to better understand the implication of the binary STDP learning rule from a machine learning point of view, studies on the equivalence to state-of-the art methods should be performed as in Hyvärinen et al. (2004) and Carlson et al. (2013). Further mathematical analysis may help us understanding better the limits and potentials of our approach in order to combine it with other approaches. The literature in machine learning and neuroscience (accurately summarized in Marblestone et al., 2016) shows that it is unlikely that only one objective function or algorithm may be responsible for all the learning capabilities of the brain. Considered combinations include supervised approach with backpropagation compatible models such as Esser et al. (2015), reinforcement learning methods (Mnih et al., 2013; Mozafari et al., 2017), as well as other unsupervised strategies such as auto-encoders and GANs.

Finally, the binary STDP along with WTA and normalization has been shown to be successful at learning in an unsupervised manner low level visual features from image data. Extension of this learning framework on temporal data is envisaged. The roles of neural oscillations in the brain are still studied, and their place in attention-demanding tasks (Dugué et al., 2015; McLelland and VanRullen, 2016) is still under debate. Nevertheless, oscillation processes like the theta-gamma model (McLelland and VanRullen, 2016) shows interesting information segmentation abilities, and may be incorporated in a network of spiking or recurrent artificial neurons (such as GRU and LTSM) as a more hard-coded WTA scheme to evaluate their impact during learning.

AUTHOR CONTRIBUTIONS

PF, FM, and ST: Designed the study; PF and FM: Analyzed the data; PF: Wrote the manuscript; PF, FM, and ST: Revised the manuscript, approved the final version, and agreed to be accountable for all aspects of the work.

FUNDING

This work was supported by the Centre National de la Recherche Scientifique (CNRS), the Agence Nationale Recherche Technologie (ANRT) and Brainchip SAS, a Brainchip Holdings Ltd company.

ACKNOWLEDGMENTS

We would like to thank Timothée Masquelier, Saeed Reza Kheradpisheh, Douglas McLelland, Christophe Garcia, and Stefan Dufner for their advice on the method and the manuscript.

REFERENCES

Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H. (2007). "Greedy layer-wise training of deep networks," in *Advances in Neural Information Processing Systems 19*, eds B. Schölkopf, J. C. Platt, T. Hoffman (Montreal, QC: MIT Press), 153–160.

Bengio, Y., Lee, D., Bornschein, J., and Lin, Z. (2015). Towards biologically plausible deep learning. *arXiv:1502.04156*.

Beyeler, M., Dutt, N. D., and Krichmar, J. L. (2013). Categorization and decision-making in a neurobiologically plausible spiking network using a STDP-like learning rule. *Neural Netw.* 48(Suppl. C), 109–124. doi: 10.1016/j.neunet.2013.07.012

Burbank, K. S. (2015). Mirrored STDP implements autoencoder learning in a network of spiking neurons. *PLoS Comput. Biol.* 11:e1004566. doi: 10.1371/journal.pcbi.1004566

Cao, Y., Chen, Y., and Khosla, D. (2015). Spiking deep convolutional neural networks for energy-efficient object recognition. *Int. J. Comp. Vis.* 113, 54–66. doi: 10.1007/s11263-014-0788-3

Carlson, K. D., Richert, M., Dutt, N., and Krichmar, J. L. (2013). "Biologically plausible models of homeostasis and stdp: stability and learning in spiking neural networks," in *Neural Networks (IJCNN), The 2013 International Joint Conference on IEEE* (Dallas, TX), 1–8.

Chan, T. H. , Jia, K., Gao, S., Lu, J., Zeng, Z., and Ma, Y. (2014). PCANet: A simple deep learning baseline for image classification. *arXiv:1404.3606*.

Chistiakova, M., Bannon, N. M., Bazhenov, M., and Volgushev, M. (2014). Heterosynaptic plasticity: multiple mechanisms and multiple roles. *Neuroscientist* 20, 483–498. doi: 10.1177/1073858414529829

Coates, A., Lee, H., and Ng, A. (2011). "An analysis of single-layer networks in unsupervised feature learning," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, Vol 15, JMLR Workshop and Conference Proceedings (JMLR W&CP)* (Fort Lauderdale, FL), 215–223.

de Almeida, L., Idiart, M., and Lisman, J. E. (2009). A second function of gamma frequency oscillations: an E%-max winner-take-all mechanism selects which cells fire. *J. Neurosci.* 29, 7497–7503. doi: 10.1523/JNEUROSCI.6044-08.2009

Delorme, A., and Thorpe, S. J. (2001). Face identification using one spike per neuron: resistance to image degradations. *Neural Netw.* 14, 795–803. doi: 10.1016/S0893-6080(01)00049-1

Deng, L., Jiao, P., Pei, J., Wu, Z., and Li, G. (2017). Gated XNOR networks: deep neural networks with ternary weights and activations under a Unified Discretization Framework. *arXiv:1705.09283*.

Diehl, P. U., and Cook, M. (2015). Unsupervised learning of digit recognition using spike-timing-dependent plasticity. *Front. Comput. Neurosci.* 9:99. doi: 10.3389/fncom.2015.00099

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

Diehl, P. U., Zarrella, G., Cassidy, A., Pedroni, B. U., and Neftci, E. (2016). Conversion of artificial recurrent neural networks to spiking neural networks for low-power neuromorphic hardware. *arXiv:1601.04187*.

Dugué, L., McLelland, D., Lajous, M., and VanRullen, R. (2015). Attention searches nonuniformly in space and in time. *Proc. Natl. Acad. Sci. U.S.A.* 112, 15214–15219. doi: 10.1073/pnas.1511331112

Esser, S. K., Appuswamy, R., Merolla, P., Arthur, J. V., and Modha, D. S. (2015). "Backpropagation for energy-efficient neuromorphic computing," in *Advances in Neural Information Processing Systems 28,* eds C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (Montreal, QC: Curran Associates, Inc.), 1117–1125.

Esser, S. K., Merolla, P. A., Arthur, J. V., Cassidy, A. S., Appuswamy, R., Andreopoulos, A., et al. (2016). Convolutional networks for fast, energy-efficient neuromorphic computing. *arXiv:1603.08270*.

Gamrat, C., Bichler, O., and Roclin, D. (2015). "Memristive based device arrays combined with spike based coding can enable efficient implementations of embedded neuromorphic circuits," in *IEEE International Electron Devices Meeting (IEDM)* (Washington, DC), 4.5.1–4.5.7.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., et al. (2014). Generative adversarial nets. in *Advances in Neural Information Processing Systems*, 2672–2680.

Goodfellow, I. J., Warde-farley, D., Mirza, M., Courville, A., and Bengio, Y. (2013). "Maxout Networks," in *ICML*, (Atlanta, GA).

Graham, B. (2014). Fractional max-pooling. *arXiv:1412.6071*.

Gupta, S., Agrawal, A., Gopalakrishnan, K., and Narayanan, P. (2015). Deep learning with Limited Numerical Precision. *arXiv:1502.02551*.

Hunsberger, E., and Eliasmith, C. (2015). Spiking deep networks with LIF neurons. *arXiv:1510.08829*.

Hyvärinen, A., Karhunen, J., and Oja, E. (2004). "Independent component analysis," *Adaptive and Cognitive Dynamic Systems: Signal Processing, Learning, Communications and Control*, ed John Wiley & Sons (Wiley-Blackwell).

Kempter, R., Gerstner, W., and van Hemmen, J. L. (2001). Intrinsic stabilization of output rates by spike-based hebbian learning. *Neural Comput.* 13, 2709–2741. doi: 10.1162/089976601317098501

Kheradpisheh, S. R., Ganjtabesh, M., Thorpe, S. J., and Masquelier, T. (2016). STDP-based spiking deep neural networks for object recognition. *arXiv:1611.01421*.

Kingma, D. P., and Welling, M. (2013). Auto-encoding variational bayes. *arXiv:1312.6114*.

Krizhevsky, A. (2009). *Learning Multiple Layers of Features from Tiny Images*. Computer Science Department, University of Toronto, Technical Report.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). "Gradient-based learning applied to document recognition," *Proc. IEEE* 86, 2278–2324. doi: 10.1109/5.726791

Leibe, B., and Schiele, B. (2003). "Analyzing appearance and con tour based methods for object categorization," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (Madison, WI), 409–415.

Makhzani, A., and Frey, B. J. (2015). "Winner-take-all autoencoders," in *Advances in Neural Information Processing Systems 28*, eds C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (Montreal, QC: MIT Press), 2791–2799.

Marblestone, A. H., Wayne, G., and Kording, K. P. (2016). Towards an integration of deep learning and neuroscience. *arXiv:1606.03813*.

Markram, H., Lübke, J., Frotscher, M., and Sakmann, B. (1997). Regulation of synaptic efficacy by coincidence of postsynaptic aps and EPSPs. *Science* 275, 213–215.

Masquelier, T., Guyonneau, R., and Thorpe, S. J. (2009). Competitive STDP-based spike pattern learning. *Neural Comput.* 21, 1259–1276. doi: 10.1162/neco.2008.06-08-804

Masquelier, T., and Thorpe, S. J. (2007). Unsupervised learning of visual features through spike timing dependent plasticity. *PLoS Comput. Biol.* 3:e31. doi: 10.1371/journal.pcbi.0030031

McLelland, D., and VanRullen, R. (2016). Theta-gamma coding meets communication-through-coherence: neuronal oscillatory multiplexing theories reconciled. *PLoS Comput. Biol.* 12:e1005162. doi: 10.1371/journal.pcbi.1005162

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *arXiv:1312.5602*.

Mozafari, M., Kheradpisheh, S. R., Masquelier, T., Nowzari-Dalini, A., and Ganjtabesh, M. (2017). First-spike based visual categorization using reward-modulated STDP. *arXiv:1705.09132*.

Nessler, B., Pfeiffer, M., and Maass, W. (2009). "STDP enables spiking neurons to detect hidden causes of their inputs," in *Advances in Neural Information Processing Systems 22*, eds Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta (Curran Associates, Inc.), 1357–1365. Available online at: http://papers.nips.cc/paper/3744-stdp-enables-spiking-neurons-to-detect-hidden-causes-of-their-inputs.pdf

Panda, P., and Roy, K. (2016). Unsupervised regenerative learning of hierarchical features in spiking deep networks for object recognition. *arXiv:1602.01510*.

Rao, R. P., and Sejnowski, T. J. (2001). Spike-timing-dependent hebbian plasticity as temporal difference learning. *Neural Comput.* 13, 2221–2237. doi: 10.1162/089976601750541787

Rasmus, A., Valpola, H., Honkala, M., Berglund, M., and Raiko, T. (2015). Semi-supervised learning with ladder network. *arXiv:1507.02672*.

Riesenhuber, M., and Poggio, T. (1999). Hierarchical models of object recognition in cortex. *Nat. Neurosci.* 2, 1019–1025.

Royer, S., and Paré, D. (2003). Conservation of total synaptic weight through balanced synaptic depression and potentiation. *Nature* 422, 518–522. doi: 10.1038/nature01530

Salimans, T., Goodfellow, I. J., Zaremba, W., Cheung, V., Radford, A., and Chen, X. (2016). Improved techniques for training gans. *arXiv:1606.03498*.

Seide, F., Fu, H., Droppo, J., Li, G., and Yu, D. (2014). "1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs," in *INTERSPEECH*, (Singapore).

Stromatias, E., Neil, D., Pfeiffer, M., Galluppi, F., Furber, S. B., and Liu, S.-C. (2015). Robustness of spiking Deep Belief Networks to noise and reduced bit precision of neuro-inspired hardware platforms. *Front. Neurosci.* 9:222. doi: 10.3389/fnins.2015.00222

Thorpe, S., Delorme, A., and Van Rullen, R. (2001). Spike-based strategies for rapid processing. *Neural Netw.* 14, 715–725. doi: 10.1016/S0893-6080(01)00083-1

Thorpe, S., Fize, D., and Marlot, C. (1996). Speed of processing in the human visual system. *Nature* 381:520.

Thorpe, S. J., Guyonneau, R., Guilbaud, N., Allegraud, J.-M., and VanRullen, R. (2004). Spikenet: real-time visual processing with one spike per neuron. *Neurocomputing* 58–60, 857–864. doi: 10.1016/j.neucom.2004.01.138

Torralba, A., Fergus, R., and Freeman, W. T. (2008). 80 million tiny images: a large data set for nonparametric object and scene recognition. *IEEE Trans. Pattern Anal. Mach. Intell.* 30, 1958–1970. doi: 10.1109/TPAMI.2008.128

Turk-Browne, N. B., Scholl, B. J., Chun, M. M., and Johnson, M. K. (2009). Neural evidence of statistical learning: efficient detection of visual regularities without awareness. *J. Cogn. Neurosci.* 21, 1934–1945. doi: 10.1162/jocn.2009.21131

Van Rullen, R., Gautrais, J., Delorme, A., and Thorpe, S. (1998). Face processing using one spike per neurone. *Biosystems* 48, 229–239.

Zhou, S., Ni, Z., Zhou, X., Wen, H., Wu, Y., and Zou, Y. (2016). Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv:1606.06160*.

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Chapter 5

Final discussion

In the state-of-the-art section 1, we have seen that both neuroscience and machine learning communities have made spectacular advances. Neuroscience has discovered many details on several brain mechanisms from cognition to physiology through extensive psychophysics and modelling studies. Machine learning, with the rapid development of Deep Learning this last decade, successfully elaborated models which are able to perform tasks with human-like accuracy, setting a new milestone in the development of artificial general intelligence. Both domains extensively use neural models which rely on heavy computations that require important hardware resources. Adapting neural networks algorithms for efficient parallel architectures is thus a critical step for accelerating the simulation of neuroscience models, as well as training deep neural networks and inference for industrial applications. The different contributions in this thesis propose several methods that can reduce the inherent costs of neural networks in terms of time and hardware resources.

In Chapter 2, the contribution is twofold. Firstly, an algorithm-architecture adequation study for the fast visual pattern detection engine BCVision, a BrainChip technology, is proposed. We have seen that the transition from a propagation scheme, which induces memory accesses penalizing performances on GPU, to a convolution scheme allows the spike-based propagation step to be ported to GPU hardware. This paradigm shift allows an overall acceleration of propagation times by a factor 6.9, although a few model changes induced a small accuracy loss. Secondly, we added complex cells to the original BCVision model both in a single network and using a coarse-to-fine hierarchy. The addition of only complex-cells reduces overall detection performance due to the loss of information by sub-sampling. However by coupling a coarse-network (which perform detection with complex cells) with a fine-network (without complex-cells) which performs detection only on pre-filtered locations, we could both

recover original performances and obtain a drastic acceleration on GPU by a factor 7000 compared to the original CPU implementation and by 233 compared to the GPU implementation.

In Chapter 3, we studied computational models of three spike-propagation algorithms for fully-connected layers with binary activations and weights on GPUs. The first algorithm processes all the positions in input and weight spaces, the second discards empty sub-packets to restrict computation to the relevant weight-matrix rows and the third references spikes and connections as lists to perform computations with an histogram algorithm. By varying the layer and hardware-specific parameters such as input-output dimensions and the sparsity in input and weight spaces, measurements of the throughputs allowed us to refine the proposed computational models through least-square optimization. With these we were able to provide ranges of parameters where each algorithm performs best, we hope this may help people interested in optimizing spike-propagation for a given network architecture and sparsity to choose the most relevant hardware design for optimal acceleration.

Finally in Chapter 4, we proposed a novel method for applying STDP to image data compliant with GPU architectures. We first showed that for non-temporal data such as images, the relative spike-time orders can be inferred with a single step convolution followed by a rectifier function. Given the intensity-latency equivalence, this rectified activation can be used with the STDP learning rule to learn low-level visual features with an unsupervised mechanism. Due to the weight-sharing in our network, and more generally in convolutional architectures, a Winner-Takes-All process based on pooling and max-filtering operations allows the selection of the most relevant regions to learn from, thus avoiding averaging biases during the computation of the update signal. Also a homeostatic process relying on post-update normalization prevents neurons from having synaptic runaway dynamics. All the proposed mechanisms are efficient on GPU. We evaluated the relevance of visual features learnt with the proposed method on four classification datasets. Performance levels were at the state-of-the-art level for unsupervised-features based classification. Thus we have shown that STDP can be integrated in convolutional neural networks while remaining biologically plausible and adapted to parallel hardware.

The different conclusions about the proposed works are now discussed along with several research perspectives.

Limits and perspectives

Rank-coding parallel implementation

Computational model of spikes propagation

We have shown in Chapter 3 three algorithms for which we proposed fine computational models. These models take into account parameters that are network-related, in particular the layer's dimensions and sparsity, as well as hardware specific. The hardware used in the benchmark, a NVidia GTX Titan X Maxwell, are proprietary and such parameters like RAS, CAS and WAS clocks are not public. We inferred a valid range for these parameters based on GDDR5 data-sheet from Micron (Micron Technology, 2014), and fine-tuned such parameters with least-square estimation. However such methods have two main flaws. First, the automatic caching effects of CUDA-compilation biases our data, while our model is based on pure memory latencies and bandwidth without cache. Second, we relied on data from a single GPU. To tackle such flaws in our study, further data should be obtained for other devices, such as other GPUs and FPGAs in order to refine our model. Nevertheless, the main aim of the study resides mainly on computational-speed trends, and our computational models should be used with this in mind before we extend it with more data.

Speed and accuracy of rank order coding networks

We managed to accelerate the spiking neural network based detection engine BCVision by adapting it for GPU architectures. Since BCVision only simulates a network with 2 layers (equivalent to V1 and V2 cells), it is able to detect visual patterns, not high-level classes of objects, since such layers can only integrate purely geometrical information. While it is theoretically possible to gather clusters involving multiple neurons (up to thousands) to discriminate objects, it is likely that having more layers will be necessary to reach the accuracy of deep learning networks while keeping reasonable processing times. Indeed without complex cells, the amount of computations per model can grow rapidly since receptive fields of about 30×30 pixels implies a local dense dot-product at each position. Choosing smaller receptive fields with sub-sampling (convolution striding or pooling) could lead to better generalization while keeping processing times low due to the intrinsic binary quantization. However such network

deepness may not be mandatory for certain visual tasks. For instance, face detection can be performed very quickly by humans (Crouzet et al., 2010) with strong hypothesis that such processes may be carried out by innate specialization (Johnson et al., 1991; Johnson and Mareschal, 2001; Ullman et al., 2012) prior to fine environmental training. Hence for certain simple tasks, relying on shallower rank-coding networks could be more relevant than deep networks if the objective is to minimize time (Perrinet and Bednar, 2015).

We produced in our first two contributions several computational studies for propagating binary spikes, both with convolution and dot-product methods. Computational efficiency of rank order coding results from two main features of such encoding : sparsity and quantized information. Developing more learning methods for rank-order networks to close gaps in performances with deep learning methods would allow to take fully advantage of their inherent sparsity and quantized information scheme. In this sense, we already mentioned in the state-of-the-art section 1.4 that many recent studies applying quantization to deep neural networks have allowed drastic acceleration with small performance loss. While these methods have only shown advantages for feedforward CNN so far, it can be expected that application to more complex architectures such as residual networks, recurrent architectures, GANs and NTM will follow in the coming years. The remaining big gap between biological and deep spike propagation is more about sparsity. Pruning and compression methods, while currently capped at 40 percent sparsity ratio (hence sixty percent of zeros in weights) without loss in accuracy and 10 percent with loss, seems a promising avenue of research in this sense. Since there is evidence that the brain may perform sparse coding for sensory information encoding (Wen et al., 2016) and associative memory (Palm, 2013), getting closer to such level of sparsity in deep neural networks could make such models far more efficient. From a computational point of view, increasing sparsity and using implementations that avoid computations with zeros can lead to very high spike throughput, as shown by our study in chapter 3. Finally, the role of spike-timing is critical in neural processes with such level of sparsity (Kloppenburg and Nawrot, 2014). Conception of neural models, such as hybrids between spiking neurons and LSTM units (Hong, 2017; Pozzi, 2018), using quantized learning and which can have high sparsity could be very efficiently implemented on parallel devices, could be trained with back-propagation. They thus may serve as a basis to understand better differences and resemblances between neuroscience and machine learning models.

Remarks on currently available hardware

While it is possible to optimize spiking neural networks propagation on synchronous devices such as CPUs, GPUs, FPGAs and ASICs, several bottlenecks and limits have been identified.

The main bottleneck identified is caused by limits in memory bandwidth and very sparse accesses to particular memory locations. As a matter of fact, during the adaptation of propagation in chapter 2 with sparse convolution and during chapter 3 with the sparse method, the sparse access pattern involved a lot of row changes, hence penalizing performance. For all the approaches, and more generally when using neural networks, reading from device memory is done extensively. The development of novel RAM memory modules such as HBM2 and GDDR6 technologies may help overcome this problem. More particularly, special modules are being developed for neuromorphic devices, including RRAM for synapse simulations and spintronics technologies for nano-scale neuron-like behaviour (Torrejon et al., 2017; Grollier et al., 2016).

The other bottleneck identified comes from operation throughputs. GPU devices (mainly NVidia) have four times less POPC units than integer and floating points additions. While the current focus on deep learning have led to optimizations for 16-bit half-floats and 8-bit integers, the novel literature on network quantization could benefit from 1 and 2-bit dedicated computational units, which may accelerate further such already fast network.

IBM's TrueNorth architecture is an example of neuromorphic architecture that tackles both problems by using distributed memory across several modules and crossbar arrays for computations. Having both quantization and management of sparse connections shows how such process can lead to fast and energy efficient neural computing. However without learning methods which can directly be applied to sparse and quantized networks, such devices are restricted to inference, while the most critical phase when using neural networks is the learning. In the case of Deep Learning, backpropagation requires an order of magnitude more complexity due to the deconvolution-like processing during backward pass. Such backpropagating signals need to be sent across all the network in a dense manner. In this sense, synthetic gradients (Jaderberg et al., 2016), which consists in learning locally the gradients to be backpropagated, could be an optimization-based response to such problems, which limit the need for the signal to be propagated across all the hierarchy. Also, better knowledge of biological learning mechanisms may help finding novel priors, like convolutions and

regularizations in the past years, in order to limit the need for backpropagation, hence keeping most of learning local to local hardware modules.

Exploring and simplifying biological priors

Influence of feedback for speed and accuracy

We proposed with our work on coarse-to-fine detection an application of fast feedback inhibition that filters information prior to its feedforward propagation to upper layers. This effectively increases sparsity, thus allowing an effective reduction in terms of number of computations, and implying acceleration by several order of magnitudes. Our coarse-to-fine detection method is currently limited to visual patterns in V2. The application of coarse-to-fine in further layer remain to be explored, both in terms of evidence in the brain and usefulness in machine learning.

More generally, oscillatory regimes of cortical circuits and feedback inhibition effects on neural processing remain to be explored. Such oscillation-based mechanisms are mostly linked to attention, synchronization, memory and learning. While the latest is well-known in machine learning as backpropagation can be seen as a feedback of reward signal (although its precise implementation in spiking circuits is still actively debated in the literature), other functions of oscillations could be integrated differently in deep neural networks.

One of the best performing architecture in computer vision, the Residual Network (ResNet), makes extensive use of residual modules, basically application of convolution and summation with previous inputs. Residual modules can be seen as a form of local feedback since the dimensionality of the inputs and outputs remain the same. Many residual modules can be applied subsequently for a given dimension, each with a different set of weights. Sharing the weights between all the residual modules at a given layer however does not harm performance, and directly makes a connection between residual networks and recurrent ones (Liao and Poggio, 2016), where time-steps are integrated explicitly by the number of modules. In addition to the ease of gradient flows in such networks, it makes perfect sense that the implementation of local excitatory and inhibitory circuits with residual modules improves the performances compared to fully feedforward architectures such as AlexNet and VGG. Nevertheless, this kind of local attention mechanism does not take into account top-down influences and yet still performs well for classification tasks. As it seems, looking at reaction

times, that humans have little need for top-down influences during more or less difficult classification tasks (only the amount of feedforward information to process may be important) (Poncet et al., 2012), the effectiveness of residual-based networks can be justified from a neuroscience view. Further studies of ResNet depth along with difficulty of classification tasks may confirm this last assumption.

Another way to implement attention in deep recurrent neural networks consists in a selection by application of a product between the activity at a given level and the softmax filtered response of this same activity (Xu et al., 2015). This attention mechanism is applied at every time-step processed by an LSTM neural network. Along with memory capacity of LSTM cells, attention mechanisms allows selection of certain features to be propagated at each time-step, hence allowing deep networks to process information sequentially. This type of attention module has been shown to beefficient for image captioning (Xu et al., 2015), neural machine translation (Luong et al., 2015) and image generation (Gregor et al., 2015). Still, such attention mechanisms are purely bottom-up since this do not make use of any information from upper layers. Looking at activities at each time step after the attention module in the context of vision, such method can effectively focus on segregated objects within the scene. This behaviour looks like what can be extracted with Theta-Gamma models of neural information filtering (McLelland and VanRullen, 2016), where local cortical inhibitory circuits and global inhibition allows extraction of salient objects.

Recent deep networks feature architectures with externalized memory modules controlled with an LSTM network. These include Neural Turing Machines (Graves et al., 2014) and Differentiable Neural Computers (Graves et al., 2016). In such architectures, softmax-controlled read and write heads allow access to an external memory space. Read-head memory in fed back into the network along with the input signal. Write-head can access and replace content of a single memory row at each time step. What is stored is however barely known. Similar processes can be observed in the medio-temporal lobe, where neurons can respond to a single concept (Quiroga et al., 2005) based on association between features. In such areas, rapid association can also be created between two or more concepts with only one presentation (Ison et al., 2015). However making a direct connection between neural networks with externalized memory and medio-temporal circuits needs to be done with caution, since there is no strong evidence that medio-temporal to V1 connections may exists to support such an hypothesis.

Local brain optimization functions

With our work on STDP with Winner-Takes-All selection, we were able to obtain biologically plausible receptive fields for V1 and V2-level cells. We have also shown the features learnt are able to reach state-of-the-art results, quantitatively and qualitatively. The method is also able to learn such features with far less examples than other methods. It have been possible to reproduce such results in a deep learning formalism by expressing STDP as a post-WTA activity maximization problem. However, there seem to be a two-layers barrier above which layer-wise training are unable too learn meaningful representation.

As proposed by Marblestone et al. (2016), studying the different brain processes in order to express them as loss functions for optimization process may help both machine learning and neuroscience communities understanding neural mechanisms. For neuroscience, bringing more mathematical background to computational neural models may help understanding the role of given cortical structures, bringing forward a normalized model of cognitive functions from which deviations can be more easily spotted and studied (in a non-prescriptive sense). From a machine learning perspective, bringing novel priors from the most advanced structure of intelligence known today to current architectures may improve drastically the proposed models. For instance, we have seen that STDP can accelerate learning by reducing the number of samples required. We hypothesize that bypassing the need to send a reward signal back to bottom layers causes this acceleration. Also, innate developmental schemes and pre-natal concepts could also guide learning in order to be faster and more efficient.

Bibliography

IBM Research: Brain-inspired Chip, Aug 2015. URL https://www.research.ibm.com/articles/brain-chip.shtml. [Online; accessed 4. Apr. 2018].

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/.

M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

L. F. Abbott. Lapicque's introduction of the integrate-and-fire model neuron (1907). *Brain research bulletin*, 50(5-6):303–304, 1999.

D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International Conference on Machine Learning*, pages 173–182, 2016.

A. Ardakani, C. Condo, and W. J. Gross. Sparsely-connected neural networks: Towards efficient vlsi implementation of deep neural networks. *arXiv preprint arXiv:1611.01427*, 2016.

M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein GAN. *arXiv preprint arXiv:1701.07875*, 2017.

M. Behrmann, J. J. Geng, and S. Shomstein. Parietal cortex and attention. *Current opinion in neurobiology*, 14(2):212–217, 2004.

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*, 7:48, 2014.

R. E. Bellman. *Adaptive control processes: a guided tour*. Princeton university press, 1961.

V. A. Bender, K. J. Bender, D. J. Brasier, and D. E. Feldman. Two coincidence detectors for spike timing-dependent plasticity in somatosensory cortex. *Journal of Neuroscience*, 26(16):4166–4177, 2006.

Y. Bengio, D. Lee, J. Bornschein, and Z. Lin. Towards biologically plausible deep learning. *arXiv*, abs/1502.04156, 2015. URL http://arxiv.org/abs/1502.04156.

J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: A cpu and gpu math compiler in python. In *Proc. 9th Python in Science Conf*, pages 1–7, 2010.

M. Beyeler, K. Carlson, T. Chou, N. C. Dutt, and J. Krichmar. A user-friendly and highly optimized library for the creation of neurobiologically detailed spiking neural networks. In *Proceedings of the International Joint Conference on Neural Networks, Killarney, Ireland*, pages 12–17, 2015.

G.-q. Bi and M.-m. Poo. Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type. *The Journal of neuroscience*, 18(24):10464–10472, 1998.

J. W. Bisley and M. E. Goldberg. The role of the parietal cortex in the neural processing of saccadic eye movements. *Advances in neurology*, 93:141–157, 2003.

R. Brette and W. Gerstner. Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *Journal of neurophysiology*, 94 (5):3637–3642, 2005.

R. Brette and D. F. Goodman. Simulating spiking neural networks on gpu. *Network: Computation in Neural Systems*, 23(4):167–182, 2012.

A. Brilhault. *Vision artificielle pour les non-voyants : une approche bio-inspirée pour la reconnaissance de formes*. PhD thesis, Université Toulouse III Paul Sabatier, Jul 2014. URL https://tel.archives-ouvertes.fr/tel-01127709.

G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016.

J. M. Budd. Extrastriate feedback to primary visual cortex in primates: a quantitative analysis of connectivity. *Proceedings of the Royal Society of London B: Biological Sciences*, 265(1400):1037–1044, 1998.

J. Bullier. Integrated model of visual processing. *Brain research reviews*, 36 (2-3):96–107, 2001.

K. S. Burbank. Mirrored STDP Implements Autoencoder Learning in a Network of Spiking Neurons. *PLoS Comput. Biol.*, 11(12):e1004566, Dec 2015. ISSN 1553-7358. doi: 10.1371/journal.pcbi.1004566.

M. Carandini and D. J. Heeger. Summation and division by neurons in primate visual cortex. *Science*, 264(5163):1333–1336, 1994.

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

T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

Z. Cheng, D. Soudry, Z. Mao, and Z. Lan. Training binary multilayer neural networks for image classification using expectation backpropagation. *arXiv preprint arXiv:1503.03562*, 2015.

K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Y. Choi, M. Choi, M. Kim, J.-W. Ha, S. Kim, and J. Choo. Stargan: Unified generative adversarial networks for multi-domain image-to-image translation. *arXiv preprint arXiv:1711.09020*, 2017.

F. Chollet et al. Keras, 2015. URL https://github.com/fchollet/keras.

R. Collobert, S. Bengio, and J. Marithoz. Torch: A modular machine learning software library, 2002.

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*, pages 3123–3131, 2015.

M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

S. M. Crouzet, H. Kirchner, and S. J. Thorpe. Fast saccades toward faces: face detection in just 100 ms. *Journal of vision*, 10(4):16–16, 2010.

G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, et al. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, 2018.

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*, 2:11, 2009.

N. Daw and H. Wyatt. Kittens reared in a unidirectional environment: evidence for a critical period. *The Journal of physiology*, 257(1):155–170, 1976.

A. Delorme and S. J. Thorpe. Face identification using one spike per neuron: resistance to image degradations. *Neural Networks*, 14(6-7):795–803, July 2001. URL https://hal.archives-ouvertes.fr/hal-00151288.

A. Delorme, L. Perrinet, and S. J. Thorpe. Networks of integrate-and-fire neurons using rank order coding b: Spike timing dependent plasticity and emergence of orientation selectivity. *Neurocomputing*, 38:539–545, 2001.

J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

L. Deng, P. Jiao, J. Pei, Z. Wu, and G. Li. Gated XNOR Networks: Deep Neural Networks with Ternary Weights and Activations under a Unified Discretization Framework. *arXiv*, May 2017. URL https://arxiv.org/abs/1705.09283.

P. U. Diehl, B. U. Pedroni, A. Cassidy, P. Merolla, E. Neftci, and G. Zarrella. Truehappiness: Neuromorphic emotion recognition on truenorth. In *Neural Networks (IJCNN), 2016 International Joint Conference on*, pages 4278–4285. IEEE, 2016a.

P. U. Diehl, G. Zarrella, A. Cassidy, B. U. Pedroni, and E. Neftci. Conversion of Artificial Recurrent Neural Networks to Spiking Neural Networks for Low-power Neuromorphic Hardware. *arXiv*, Jan 2016b. URL https://arxiv.org/abs/1601.04187.

S. Dreyfus. The numerical solution of variational problems. *Journal of Mathematical Analysis and Applications*, 5(1):30–45, 1962.

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. *Proc. Natl. Acad. Sci. U.S.A.*, 113(41):11441–11446, Oct 2016. ISSN 0027-8424. doi: 10.1073/pnas.1604850113.

M. A. Farries and A. L. Fairhall. Reinforcement learning with modulated spike timing–dependent synaptic plasticity. *Journal of neurophysiology*, 98 (6):3648–3665, 2007.

R. Fergus, P. Perona, and A. Zisserman. A sparse object category model for efficient learning and exhaustive recognition. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 380–387. IEEE, 2005.

A. K. Fidjeland and M. P. Shanahan. Accelerated simulation of spiking neural networks using gpus. In *Neural Networks (IJCNN), The 2010 International Joint Conference on*, pages 1–8. IEEE, 2010.

I. M. Finn and D. Ferster. Computational diversity in complex cells of cat primary visual cortex. *Journal of Neuroscience*, 27(36):9638–9648, 2007.

D. B. Fogel. *Evolutionary computation: the fossil record*. Wiley-IEEE Press, 1998.

L. J. Fogel, A. J. Owens, and M. J. Walsh. Artificial intelligence through simulated evolution. 1966.

K. Fukushima. A hierarchical neural network model for selective attention. In *Neural computers*, pages 81–90. Springer, 1989.

K. Fukushima and S. Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*, pages 267–285. Springer, 1982.

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

T. Gao, D. Harari, J. Tenenbaum, and S. Ullman. When computer vision gazes at cognition. *arXiv preprint arXiv:1412.2672*, 2014.

K. Glebov, M. Löchner, R. Jabs, T. Lau, O. Merkel, P. Schloss, C. Steinhäuser, and J. Walter. Serotonin stimulates secretion of exosomes from microglia cells. *Glia*, 63(4):626–634, 2015.

V. Goffaux, J. Peters, J. Haubrechts, C. Schiltz, B. Jansma, and R. Goebel. From coarse to fine? spatial and temporal dynamics of cortical face processing. *Cerebral Cortex*, 21(2):467–476, 2010.

I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

D. F. Goodman and R. Brette. The brian simulator. *Frontiers in neuroscience*, 3(2):192, 2009.

P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch SGD: training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

GPUEater. Benchmark cifar10 on tensorflow with rocm on amd gpus vs cuda9 and cudnn7 on nvidia gpus, 2018. URL http://blog.gpueater.com/en/2018/04/23/00011_tech_cifar10_bench_on_tf13/.

A. Graves, G. Wayne, and I. Danihelka. Neural Turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471, 2016.

K. Gregor, I. Danihelka, A. Graves, D. J. Rezende, and D. Wierstra. Draw: A recurrent neural network for image generation. *arXiv preprint arXiv:1502.04623*, 2015.

J. Grollier, D. Querlioz, and M. D. Stiles. Spintronic nanodevices for bioinspired computing. *Proceedings of the IEEE*, 104(10):2024–2039, 2016.

S. Gu, T. Lillicrap, I. Sutskever, and S. Levine. Continuous deep q-learning with model-based acceleration. In *International Conference on Machine Learning*, pages 2829–2838, 2016.

J. Guerguiev, T. P. Lillicrap, and B. A. Richards. Towards deep learning with segregated dendrites. *eLife*, 6, 2017.

I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. C. Courville. Improved training of wasserstein gans. In *Advances in Neural Information Processing Systems*, pages 5769–5779, 2017.

R. Guyonneau, R. VanRullen, and S. J. Thorpe. Neurons tune to the earliest spikes through stdp. *Neural Computation*, 17(4):859–879, 2005.

R. H. Hahnloser, R. Sarpeshkar, M. A. Mahowald, R. J. Douglas, and H. S. Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405(6789):947, 2000.

J. K. Hamlin. The case for social evaluation in preverbal infants: gazing toward one's goal drives infants' preferences for helpers over hinderers in the hill paradigm. *Frontiers in psychology*, 5:1563, 2015.

S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, et al. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

hindupuravinash. The gan zoo, Mar 2018. URL https://github.com/hindupuravinash/the-gan-zoo. [Online; accessed 30. Mar. 2018].

M. L. Hines, T. Morse, M. Migliore, N. T. Carnevale, and G. M. Shepherd. Modeldb: a database to support computational neuroscience. *Journal of computational neuroscience*, 17(1):7–11, 2004.

R. V. Hoang, D. Tanna, L. C. Jayet Bray, S. M. Dascalu, and F. C. Harris Jr. A novel CPU/GPU simulation environment for large-scale biologically realistic neural modeling. *Frontiers in neuroinformatics*, 7:19, 2013.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

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*, 117(4):500–544, 1952.

C. Hong. Training spiking neural networks for cognitive tasks: A versatile framework compatible to various temporal codes. *arXiv preprint arXiv:1709.00583*, 2017.

K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

X. Huang and S. Belongie. Arbitrary style transfer in real-time with adaptive instance normalization. *CoRR, abs/1703.06868*, 2017.

D. H. Hubel and T. N. Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106–154, 1962.

S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros. Image-to-image translation with conditional adversarial networks. *CVPR*, 2017.

M. J. Ison, R. Q. Quiroga, and I. Fried. Rapid encoding of new memories by individual neurons in the human brain. *Neuron*, 87(1):220–230, 2015.

E. M. Izhikevich. Simple model of spiking neurons. *IEEE Transactions on neural networks*, 14(6):1569–1572, 2003.

E. M. Izhikevich. Solving the distal reward problem through linkage of stdp and dopamine signaling. *Cerebral cortex*, 17(10):2443–2452, 2007.

M. Jaderberg, W. M. Czarnecki, S. Osindero, O. Vinyals, A. Graves, D. Silver, and K. Kavukcuoglu. Decoupled neural interfaces using synthetic gradients. *arXiv preprint arXiv:1608.05343*, 2016.

Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature

embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

M. H. Johnson and D. Mareschal. Cognitive and perceptual development during infancy. *Current Opinion in Neurobiology*, 11(2):213–218, 2001.

M. H. Johnson, S. Dziurawiec, H. Ellis, and J. Morton. Newborns' preferential tracking of face-like stimuli and its subsequent decline. *Cognition*, 40(1-2): 1–19, 1991.

N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12. ACM, 2017.

U. R. Karmarkar and D. V. Buonomano. A model of spike-timing dependent plasticity: one or two coincidence detectors? *Journal of Neurophysiology*, 88 (1):507–513, 2002.

T. Karras, T. Aila, S. Laine, and J. Lehtinen. Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*, 2017.

N. Ketkar. Introduction to pytorch. In *Deep Learning with Python*, pages 195–208. Springer, 2017.

S. R. Kheradpisheh, M. Ganjtabesh, S. J. Thorpe, and T. Masquelier. Stdp-based spiking deep neural networks for object recognition. *arXiv*, abs/1611.01421, 2016. URL http://arxiv.org/abs/1611.01421.

B. Kisačanin. Deep learning for autonomous vehicles. In *Multiple-Valued Logic (ISMVL), 2017 IEEE 47th International Symposium on*, pages 142–142. IEEE, 2017.

P. Kloppenburg and M. P. Nawrot. Neural coding: sparse but on time. *Current Biology*, 24(19):R957–R959, 2014.

N. Kodali, J. Hays, J. Abernethy, and Z. Kira. On convergence and stability of gans. 2018.

A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

A. Krogh and J. A. Hertz. A simple weight decay can improve generalization. In *Advances in neural information processing systems*, pages 950–957, 1992.

S. Kunkel, T. C. Potjans, J. M. Eppler, H. E. E. Plesser, A. Morrison, and M. Diesmann. Meeting the memory challenges of brain-scale network simulation. *Frontiers in neuroinformatics*, 5:35, 2012.

X. Lagorce, E. Stromatias, F. Galluppi, L. A. Plana, S.-C. Liu, S. B. Furber, and R. B. Benosman. Breaking the millisecond barrier on SpiNNaker: implementing asynchronous event-based plastic models with microsecond resolution. *Front. Neurosci.*, 9, Jun 2015. ISSN 1662-453X. doi: 10.3389/fnins.2015.00206.

A. B. L. Larsen, S. K. Sønderby, H. Larochelle, and O. Winther. Autoencoding beyond pixels using a learned similarity metric. *arXiv preprint arXiv:1512.09300*, 2015.

Y. LeCun, D. Touresky, G. Hinton, and T. Sejnowski. A theoretical framework for back-propagation. In *Proceedings of the 1988 connectionist models summer school*, pages 21–28. CMU, Pittsburgh, Pa: Morgan Kaufmann, 1988.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, et al. Photo-realistic single image super-resolution using a generative adversarial network. *arXiv preprint*, 2016.

F.-F. Li, M. Andreetto, and M. A. Ranzato. Caltech101 image dataset. 2003. URL http://www.vision.caltech.edu/Image_Datasets/Caltech101/.

Y. Li, N. Wang, J. Shi, J. Liu, and X. Hou. Revisiting batch normalization for practical domain adaptation. *arXiv preprint arXiv:1603.04779*, 2016.

Q. Liao and T. Poggio. Bridging the gaps between residual learning, recurrent neural networks and visual cortex. *arXiv preprint arXiv:1604.03640*, 2016.

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

S. Linnainmaa. The representation of the cumulative rounding error of an algorithm as a taylor expansion of the local rounding errors. *Master's Thesis (in Finnish), Univ. Helsinki*, pages 6–7, 1970.

B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Pensky. Sparse convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 806–814, 2015.

M.-T. Luong, H. Pham, and C. D. Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

A. H. Marblestone, G. Wayne, and K. P. Kording. Toward an integration of deep learning and neuroscience. *Frontiers in Computational Neuroscience*, 10:94, 2016.

H. Markram. The human brain project. *Scientific American*, 306(6):50–55, 2012.

H. Markram, J. Lübke, M. Frotscher, and B. Sakmann. Regulation of synaptic efficacy by coincidence of postsynaptic aps and epsps. *Science*, 275(5297): 213–215, 1997. ISSN 0036-8075. URL http://science.sciencemag.org/content/275/5297/213.

T. Masquelier. Stdp allows close-to-optimal spatiotemporal spike pattern detection by single coincidence detector neurons. *Neuroscience*, 2017.

T. Masquelier and S. J. Thorpe. Unsupervised Learning of Visual Features through Spike Timing Dependent Plasticity. *PLoS Computational Biology*, 3(2):e31, Feb. 2007. URL https://hal.archives-ouvertes.fr/hal-00135582.

W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5:115–133, 1943.

D. McLelland and R. VanRullen. Theta-Gamma Coding Meets Communication-through-Coherence: Neuronal Oscillatory Multiplexing Theories Reconciled. *PLoS Comput Biol*, 12(10):e1005162+, Oct. 2016. URL http://dx.doi.org/10.1371/journal.pcbi.1005162.

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, et al. A million

spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014.

I. Micron Technology. Gddr5 sgram edw4032babg datasheet, 2014. URL https://www.micron.com/parts/dram/gddr5/edw4032babg-70-f.

M. Mikaitis, G. Pineda García, J. C. Knight, and S. B. Furber. Neuromodulated synaptic plasticity on the spinnaker neuromorphic system. *Frontiers in neuroscience*, 12:105, 2018.

T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013a.

T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013b.

M. Mishkin, L. G. Ungerleider, and K. A. Macko. Object vision and spatial vision: two cortical pathways. *Trends in neurosciences*, 6:414–417, 1983.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz. Pruning convolutional neural networks for resource efficient inference. 2016.

P. Moscato et al. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. *Caltech concurrent computation program, C3P Report*, 826:1989, 1989.

M. Mozafari, S. R. Kheradpisheh, T. Masquelier, A. Nowzari-Dalini, and M. Ganjtabesh. First-spike based visual categorization using reward-modulated STDP. *arXiv*, May 2017. URL https://arxiv.org/abs/1705.09132.

V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, and D. Marr. Accelerating binarized neural networks: comparison of fpga, cpu, gpu, and asic. In *Field-Programmable Technology (FPT), 2016 International Conference on*, pages 77–84. IEEE, 2016.

NVIDIA Corporation. NVIDIA CUDA C programming guide, 2010. URL http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

G. Palm. Neural associative memories and sparse coding. *Neural Networks*, 37: 165–171, 2013.

L. Perrinet. Emergence of filters from natural scenes in a sparse spike coding scheme. *Neurocomputing*, 58:821–826, 2004.

L. Perrinet and M. Samuelides. Coherence detection in a spiking neuron via hebbian learning. *Neurocomputing*, 44:133–139, 2002.

L. Perrinet, M. Samuelides, and S. Thorpe. Sparse spike coding in an asynchronous feed-forward multi-layer neural network using matching pursuit. *Neurocomputing*, 57:125–134, 2004.

L. U. Perrinet. Role of homeostasis in learning sparse representations. *Neural computation*, 22(7):1812–1836, 2010.

L. U. Perrinet and J. A. Bednar. Edge co-occurrences can account for rapid categorization of natural versus animal images. *Scientific reports*, 5:11400, 2015.

T. Poggio and E. Bizzi. Generalization in vision and motor control. *Nature*, 431(7010):768, 2004.

M. Poncet, L. Reddy, and M. Fabre-Thorpe. A need for more information uptake but not focused attention to access basic-level representations. *Journal of vision*, 12(1):15–15, 2012.

I. Pozzi. Developing a spiking neural model of long short-term memory architectures. 2018.

R. Q. Quiroga, L. Reddy, G. Kreiman, C. Koch, and I. Fried. Invariant visual representation by single neurons in the human brain. *Nature*, 435(7045):1102, 2005.

A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv*, abs/1511.06434, 2015. URL http://arxiv.org/abs/1511.06434.

M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

I. Rechenberg. Cybernetic solution path of an experimental problem. 1965.

S. Reed, Z. Akata, X. Yan, L. Logeswaran, B. Schiele, and H. Lee. Generative adversarial text to image synthesis. *arXiv preprint arXiv:1605.05396*, 2016.

A. Reynaud, G. S. Masson, and F. Chavane. Dynamics of local input normalization result from balanced short-and long-range intracortical interactions in area v1. *Journal of neuroscience*, 32(36):12558–12569, 2012.

M. Riesenhuber and T. Poggio. Hierarchical models of object recognition in cortex. *Nature neuroscience*, 2(11):1019–1025, 1999.

F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. 65:386 – 408, 12 1958.

G. A. Rousselet, S. J. Thorpe, and M. Fabre-Thorpe. Taking the max from neuronal responses. *Trends in cognitive sciences*, 7(3):99–102, 2003.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533, 1986.

N. C. Rust, V. Mante, E. P. Simoncelli, and J. A. Movshon. How MT cells analyze the motion of visual patterns. *Nature neuroscience*, 9(11):1421, 2006.

T. Salimans, I. J. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen. Improved techniques for training gans. *arXiv*, abs/1606.03498, 2016. URL http://arxiv.org/abs/1606.03498.

T. Salimans, J. Ho, X. Chen, and I. Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

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

F. Seide and A. Agarwal. Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2135–2135. ACM, 2016.

F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

T. Serre, A. Oliva, and T. Poggio. A feedforward architecture accounts for rapid categorization. *Proceedings of the national academy of sciences*, 104 (15):6424–6429, 2007.

D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529 (7587):484–489, 2016.

A. E. Skerry and E. S. Spelke. Preverbal infants identify emotional reactions that are incongruent with goal outcomes. *Cognition*, 130(2):204–216, 2014.

D. Soudry, I. Hubara, and R. Meir. Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights. In *Advances in Neural Information Processing Systems*, pages 963–971, 2014.

M. Spratling. Cortical region interactions and the functional role of apical dendrites. *Behavioral and cognitive neuroscience reviews*, 1(3):219–228, 2002.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

I. Sugiarto, G. Liu, S. Davidson, L. A. Plana, and S. B. Furber. High performance computing on spinnaker neuromorphic platform: a case study for energy efficient image processing. In *Performance Computing and Communications Conference (IPCCC), 2016 IEEE 35th International*, pages 1–8. IEEE, 2016.

C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, et al. Going deeper with convolutions. Cvpr, 2015.

C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, volume 4, page 12, 2017.

Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL http://arxiv.org/abs/1605.02688.

S. Thorpe, D. Fize, and C. Marlot. Speed of processing in the human visual system. *nature*, 381(6582):520, 1996.

S. Thorpe, A. Delorme, and R. Van Rullen. Spike-based strategies for rapid processing. *Neural networks*, 14(6):715–725, 2001.

S. Thorpe, A. Yousefzadeh, J. Martin, and T. Masquelier. Unsupervised learning of repeating patterns using a novel stdp based algorithm. *Journal of Vision*, 17(10):1079–1079, 2017.

S. J. Thorpe, R. Guyonneau, N. Guilbaud, J.-M. Allegraud, and R. VanRullen. Spikenet: real-time visual processing with one spike per neuron. *Neurocomputing*, pages 857 – 864, 2004. ISSN 0925-2312. URL //www.sciencedirect.com/science/article/pii/S0925231204001432. Computational Neuroscience: Trends in Research 2004.

R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

S. B. Tieman and H. V. Hirsch. Exposure to lines of only one orientation modifies dendritic morphology of cells in the visual cortex of the cat. *Journal of Comparative Neurology*, 211(4):353–362, 1982.

J. Togelius, S. M. Lucas, and R. De Nardi. Computational intelligence in racing games. In *Advanced Intelligent Paradigms in Computer Games*, pages 39–69. Springer, 2007.

J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, 2011.

J. Torrejon, M. Riou, F. A. Araujo, S. Tsunegi, G. Khalsa, D. Querlioz, P. Bortolotti, V. Cros, K. Yakushiji, A. Fukushima, et al. Neuromorphic computing with nanoscale spintronic oscillators. *Nature*, 547(7664):428, 2017.

S. Ullman, D. Harari, and N. Dorfman. From simple innate biases to complex visual concepts. *Proceedings of the National Academy of Sciences*, 109(44): 18215–18220, 2012.

V. K. Valsalam, J. Hiller, R. MacCurdy, H. Lipson, and R. Miikkulainen. Constructing controllers for physical multilegged robots using the enso neuroevolution approach. *Evolutionary Intelligence*, 5(1):45–56, 2012.

A. Van Den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

R. Van Rullen, J. Gautrais, A. Delorme, and S. Thorpe. Face processing using one spike per neurone. *Biosystems*, 48(1):229–239, 1998.

V. N. Vapnik. An overview of statistical learning theory. *IEEE transactions on neural networks*, 10(5):988–999, 1999.

L. N. Vaserstein. Markov processes over denumerable products of spaces, describing large systems of automata. *Problemy Peredachi Informatsii*, 5(3): 64–72, 1969.

R. Watt. Scanning from coarse to fine spatial scales in the human visual system after the onset of a stimulus. *JOSA A*, 4(10):2006–2021, 1987.

W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems*, pages 2074–2082, 2016.

P. J. Werbos. Applications of advances in nonlinear sensitivity analysis. In *System modeling and optimization*, pages 762–770. Springer, 1982.

S. Wiesler, R. Schlüter, and H. Ney. A convergence analysis of log-linear training and its application to speech recognition. In *Automatic Speech Recognition and Understanding (ASRU), 2011 IEEE Workshop on*, pages 1–6. IEEE, 2011.

M. A. Wilson, U. S. Bhalla, J. D. Uhley, and J. M. Bower. Genesis: A system for simulating neural networks. In *Advances in neural information processing systems*, pages 485–492, 1989.

K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine Learning*, pages 2048–2057, 2015.

E. Yavuz, J. Turner, and T. Nowotny. Simulating spiking neural networks on massively parallel graphical processing units using a code generation approach with genn. *BMC neuroscience*, 15(1):O1, 2014.

Y. Zhang, W. Chan, and N. Jaitly. Very deep convolutional networks for end-to-end speech recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on*, pages 4845–4849. IEEE, 2017.

S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

C. Zhu, S. Han, H. Mao, and W. J. Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.

J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networkss. In *Computer Vision (ICCV), 2017 IEEE International Conference on*, 2017.