POLITECNICO DI TORINO

M.Sc.'s Degree in Electronic Engineering

M.Sc.'s Degree Thesis

Efficient Implementation of Spiking Neural Networks on the Loihi Neuromorphic Processor

Supervisors

Prof. Maurizio MARTINA

Prof. Muhammad SHAFIQUE

Project Ass. Alberto MARCHISIO

Candidate

Riccardo MASSA

03 2020

Summary

Recent developments of deep **Artificial Neural Networks** (ANNs) have pushed forward the state-of-the-art in the field of image recognition [1]. However, the high power demand required by these networks when it comes to perform inference tasks on edge devices limits the spread of ANNs in context where the energy/power consumption is crucial. On the other hand, **Spiking Neural Networks** (SNNs), due to their biologically inspired behavior, have shown promising results both in terms of power efficiency and real-time classification performance [2]. Being the comunication between neurons based on spikes, SNNs guarantee a lower computational load, as well as a reduction of latency.
Along with the development of efficient SNN specialized accelerators (TrueNorth [3], SpiNNaker [4] and **Intel Loihi** [5]), another advancement in the field of neuromorphic hardware has come from a new generation of camera, the **DVS event-based sensor** [6]. Such a device, differently from a classical frame-based camera, works emulating the behavior of the human retina. Thus, the recorded information is not a series of time-wise separated frames, but a sequence of spikes, which are generated every time a change of light intensity is detected. The event-based behavior of these sensors pairs well with SNNs: the output of a DVS camera can be used as input of the SNN, which collects and elaborate events in real-time.
A promising approach to train SNNs in a supervised learning scenario is to train an ANN with state-of-the-art backpropagation approaches, and then assign the trained parameters (weights and biases) to an equivalent SNN applying a conversion process. This approach has shown promising results [7], mostly because it allows to get the best from the two worlds: the converted SNN totally behaves like a normal SNN, with its benefits in terms of efficiency and latency. At the same time, the network has been trained in ANN domain with high performing methodologies that ensure good results in classification tasks. However, such a conversion may not always held the expected results. In fact, many aspects has to be taken into account, like the original ANN structure, the training process, as well as the parameters that control the ANN-to-SNN conversion. This is especially true when the converted SNN has to be deployed on a limited precision hardware like Intel Loihi, which restricts the degree of freedom of the conversion process.

For this reason, in this thesis we present a complete **ANN-to-SNN design process**, systematically discussing the effects of the main parameters that take part in the conversion. We evaluate their effect, and extract some general rules that can be successfully applied when it comes to develop an SNN for Intel Loihi. Once we have an SNN that gets good accuracy results both on the MNIST [8] and the CIFAR10 [9] datasets, we evaluate it also on the DVS gesture dataset [10], which comprise 11 gestures recorded with a DVS event-based camera. The main challenge when adopting the ANN-to-SNN conversion approach to get a trained SNN is that we can not train an ANN on the event series coming from the DVS camera. For this reason, we first need to collect the events into frames and then train the ANN on such converted dataset. Different pre-processing techniques are discussed in this thesis, also reporting the accuracy results achieved by the ANN on the generated converted dataset. Then, after the conversion, the SNN is tested on the DVS Gesture dataset, and it is ready to be deployed for real-time classification on Intel Loihi.
Finally, we have optimized the conversion of a hardware-efficient ANN, the **MobileNet** [11], into its equivalent SNN. The network has been designed in order to achieve the best accuracy results while minimizing the hardware resources of the Loihi processor.

Acknowledgements

I want to express my gratitude to Professor Martina, who gave me the possibility to have this great experience in Wien.
I am grateful to Professor Shafique, that guided me through this thesis with his knowledge and dedication. I want also to thank Alberto, who have followed me in this project helping me with his experience and kindness.
Thanks Antonio, Valerio and Giuseppe, it wouldn't have been the same without you guys.
To Serena, for being the light that has never gone out.
To my family, for their support throughout all these years. They have always been my safe place, and my source of happiness.

Table of Contents

List of Tables

List of Figures

Acronyms

AI: Artificial Intelligence

ML: Machine Learning

ANN: Artificial Neural Networks

CNN: Convolutional Neural Networks

SNN: Spiking Neural Networks

DVS: Dynamic Vision Sensor

WR: Weight Regularization

DThIR: Desired Threshold to Input Ratio

MSE: Mean Square Error

SGD: Stochastic Gradient Descent

LIF: Leaky Integrate and Fire

CUBA: Current Based

ISI: Inter-Spike Interval

TTFS: Time to First Spike

STDP: Spike Time Dependent Plasticity

NoC: Network on Chip

ReLU: Rectified Linear Unit

DW: Depthwise

PC: Pearson Coefficient

Chapter 1

Introduction

1.1 Motivation

Recent developments in the field of Deep Learning have pushed forward the limits of artificial intelligence in several scientific fields. Artificial Neural Networks (ANNs) have become a widely adopted solution to solve many highly non-linear classification and regression problems, frequently overcoming the ability of the human brain in such tasks [12]. Improvements in the cognitive capabilities of artificial intelligence systems are allowing big steps forward in fields like autonomous driving, healtcare, and even finance and marketing [13] [14] [15] [16].
However, there is still a huge difference in terms of *power efficiency* between modern computers and the human brain. To achieve the same computational performance of the brain, a supercomputer needs several orders of magnitude higher power [17]. This is due to the fundamental differences between how the computation is performed in a biological brain and in a classical Von Neumann computer architecture. Concerning artificial intelligence, some specific hardware architectures have been proposed as ANNs accelerators, like Google TPU [18]. However, the computation that take place in conventional hardware designs is far from the intrinsic efficiency that takes place in the human brain. This large gap in terms of power efficiency has given rise to the question: *can we build brain-inspired computer systems for highly efficient computation?*
To give a positive answer to this challenging question, we need to radically change the way computation is performed, both working on the software and hardware levels.
In the past few years Spiking Neural Networks (SNNs) have emerged as the most energy efficient type of Neural networks. These networks, differently from ANNs, base their computation on faithful biological models of neurons, adopting a spike-based communication that leads to a low-energy, real-time computation [19].

Because of their bio-inspired computation, these networks are the most promising solution to reduce the gap with the human brain, both in terms of power efficiency and real-time use-cases. However, in order to release their full potential, SNNs need custom hardware that can execute neuron computation in an efficient way.
Neuromorphic computing, a novel field in computer architecture design, has achieved huge steps forward in the development of brain-inspired computer architectures, specifically designed to run SNNs. The core idea is to reproduce on the hardware level the way the neurons execute computations in the brain. This translates into the development of highly parallel architectures in which each processing unit reproduces the behavior of a neuron, and neurons can communicate through spikes. This radical shift in the hardware design promises huge benefits in terms of power consumption. Examples of such neuromorphic designs are IBM TrueNorth [3], SpiNNaker [4], BrainScale [20] and Intel Loihi [5]. Our research has focused on the latter platform. Figure 1.1 shows a comparison of several hardware architectures, showing how competitive neuromorphic designs can be when compared to conventional hardware solutions in terms of power consumption.
Along the line of brain-inspired technologies, a novelty also comes from vision sensor field. Event-based cameras [6] are vision sensors that mimic the behavior of the human retina, and generate spikes with the movements of the recorded subject. Thanks to their bio-inspired neuromorphic design, these sensors can coop well with SNNs.
However, because of the fact that neuromorphic chips are relatively new, a lot of effort still needs to be put in the development of compatible SNNs that can fully exploit the potential of these devices. Moreover, even if SNNs have shown great results in terms of power efficiency and real-time behavior, they still do not provide the same accuracy that can be achieved with state-of-the-art ANNs.

1.2 Scientific challenges

As a consequence of the intrinsic differences between conventional ANNs and SNNs, the training procedures that are used to train ANNs to reach high levels of accuracy can not be applied directly to SNNs. This entails a critical issue to the development of deep spiking networks able to achieve the same accuracy of their artificial counterpart. A possible training technique that have shown promising results consists in training conventional ANNs with state-of-the-art algorithms, and then convert the ANN into an equivalent SNN.
This solution, however, implies some limitations that are related to the key differences between ANN and SNN models. Therefore, to achieve a good equivalent SNN it is necessary to find the right balance for all the parameters implied in the conversion, as well as efficiently translate the parameters learned during the ANN

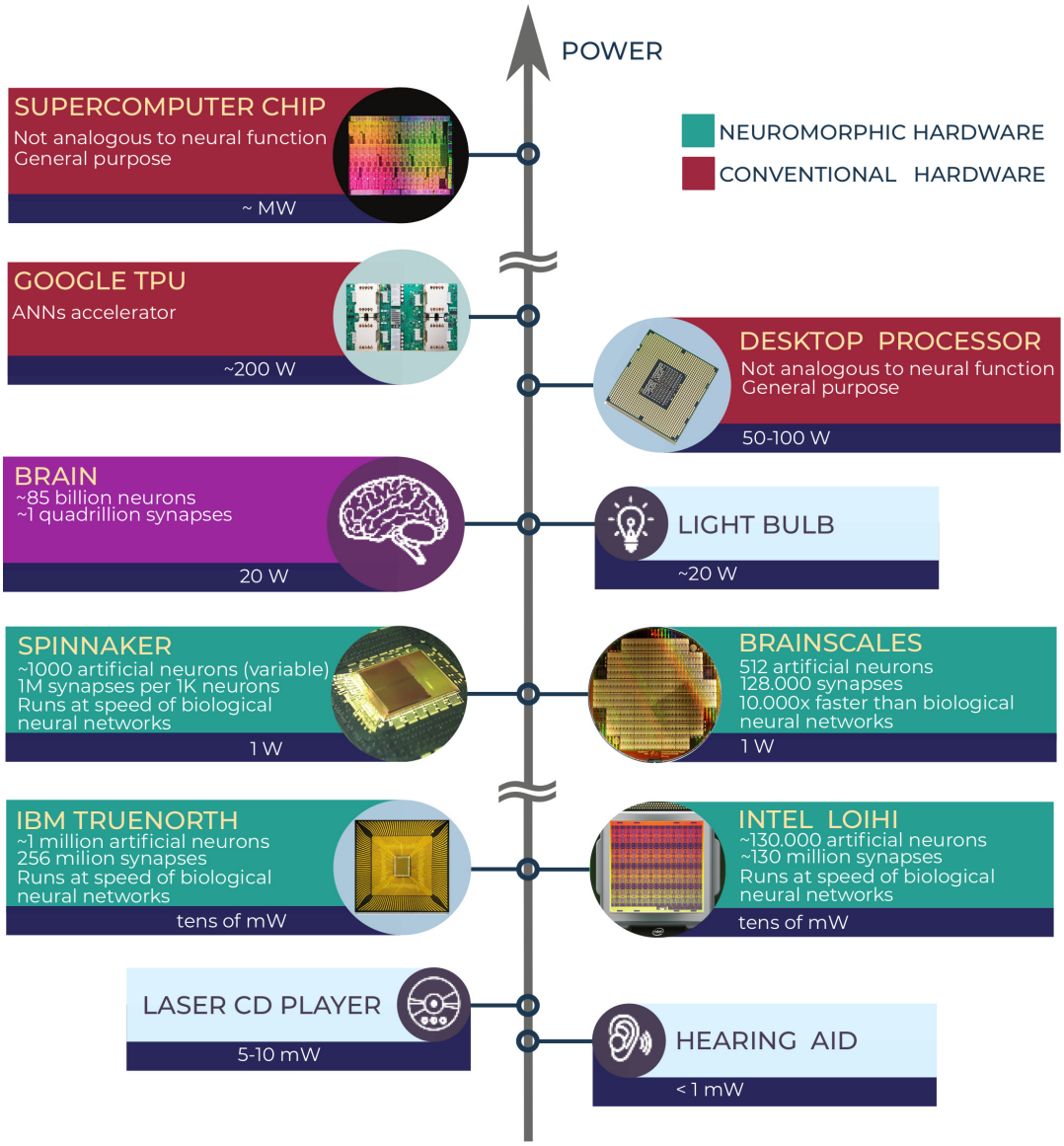


Figure 1.1: Comparison of power consumption between conventional hardware architectures and neuromorphic architectures [Source: [21]]

training into equivalent parameters for the SNN.
Moreover, building SNNs that can be executed on the Loihi neuromorphic chip requires to take into account several constraints that are specifically hardware related such as quantized synaptic weights and biases, and may consistently limit the final result.
Regarding the use of event-based cameras in combination with SNNs, it is necessary

to find an optimized conversion policy in order to make the events collected by the sensor compatible with the frame-based training executed in the ANN domain.

1.3 Novel contributions

In this research, we have focused on the optimization of the ANNs-to-SNNs conversion process, in order to build efficient spiking networks. Our study is specifically related to the Loihi neuromorphic chip, and therefore the optimizations require to take into account several hardware limitations that influence the final result of the conversion.
Moreover, we have proposed a conversion technique for converting an event-based gesture dataset, into an equivalent frame-based dataset in order to be compatible with the ANN training. Then, we have trained the same ANN optimized in the previous research step and we have efficiently converted it into an equivalent SNN for recognizing gestures on Loihi.
Finally, we have optimized the conversion of a hardware-efficient ANN, the MobileNet [11], into an equivalent SNN. The network has been designed in order to achieve the best accuracy results while minimizing the hardware resources of the Loihi processor. The thesis is organized as follow:

- **Chapter 2**: we present an overview of the basic notions of machine learning, artificial and spiking neural networks, as well as a description of the Loihi architecture and event-based cameras.
- **Chapter 3**: we perform a comprehensive analysis of the setup for converting an ANN into a SNN, finding the optimal parameters for a correct conversion of a convolutional ANN on the Loihi platform.
- **Chapter 4**: We design a pre-processing method for a gesture event-based dataset, to produce a frame-based equivalent dataset compatible with the ANN domain. Then, we train the same optimized ANN of Chapter 3 on the pre-processed dataset and convert it into a corresponding SNN to be ready to be deployed on Intel Loihi.
- **Chapter 5**: We propose an in-depth analysis of a MobileNet ANN-to-SNN conversion in order to build an equivalent spiking MobileNet that achieves the highest accuracy possible while limiting the hardware occupation of the Loihi processor.
- **Chapter 6**: We draw the conclusions of our work.

A schematic view of chapters 3,4 and 5 is presented in Figure 1.2.

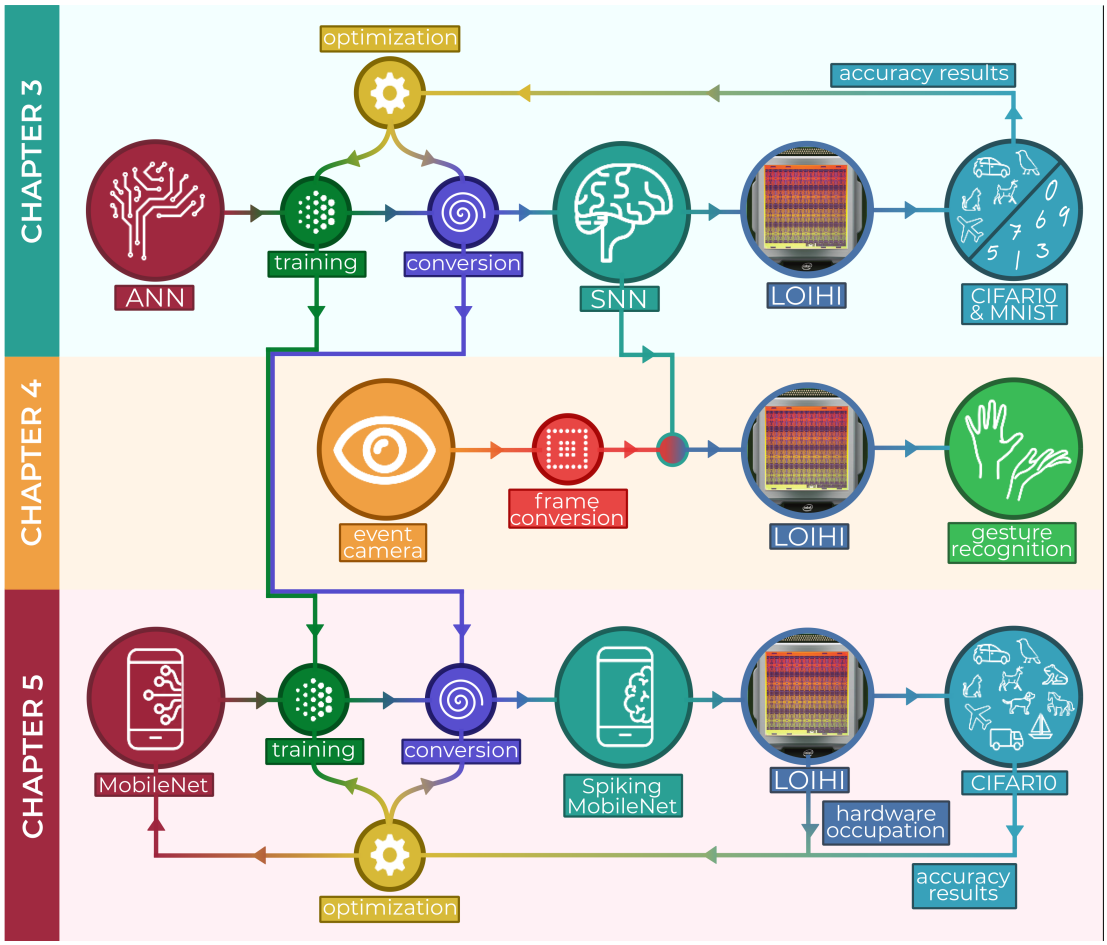


Figure 1.2: Thesis structure and overview of the experimental flow.

Chapter 2

Background

In this chapter are discussed the basic concepts that will be the fundamental of the research. Starting form the concepts of Artificial Intelligence and Deep Learning, we will move towards Spiking Neural Networks, with a final overview of Intel Loihi and DVS camera.

2.1 Artificial Intelligence and Machine Learning

Artificial Intelligence (**AI**) is a scientific field that aims to create machines able to solve problems mimicking the human cognition. These problems can be easily described using a formal mathematical approach, and are hardly solvable by the human brain because of the highly intensive computation required.
However, AI algorithms may be of little use when it comes to solve more intuitive problems, that can hardly be described formally. Usually these problems are easily tackled by the human brain, thanks to its ability to generalize and use intuition.
As a consequence, a subset of AI called **Machine Learning** (**ML**) has been developed in recent years to overcome this limitation. ML algorithms find the statistical relations among a series of inputs. In order to do so, the inputs need to be pre-processed to extract a high level representation of the data, called features. The algorithms are then trained on these features, learning how to correctly process the incoming inputs [22].

2.2 Neural Networks and Deep Learning

Machine learning aims to build biologically inspired algorithms, capable of emulate the interactions between neurons that take place in the brain and that are fundamental for the human cognition. Neural Networks (NN) are biologically inspired algorithms that try to replicate the basic step of information processing

that take place in the brain. Based on a mathematical representation of neurons and synaptic communication, they elaborate the inputs information in order to solve a classification or a regression problem.
The basic structure of a NN usually makes use of a low number of neurons, arranged into a maximum of 3 layers: input, inner (or *hidden*) and output layers. Neurons are connected in a all-to-all fashion (*fully connected*), and the output neurons generate the prediction based on the inputs received from the input layer.
In order to build more and more complex and intelligent systems, the number of inner layers has been increased getting to a new step in the machine learning evolution, Deep Learning.
The presence of an higher number of hidden layers allows the network to extract features all by itself, directly starting from the incoming inputs. These features contain a progressively higher level representation of the inputs [22].
As a case of study, let's consider the problem of classification of a sequence of images. The human brain is capable of recognizing the subjects, backgrounds and all the different elements in a given picture thanks to the process of hierarchical visual feature representation that take place in the brain. As the visual information gets processed by the visual cortex, higher features are extracted and combined to create more complex features, until a clear representation of the input image is obtained.
Following this approach, the same hierarchical feature extraction is performed in a neural network. Layer after layer, higher level features are extracted and combined into more complex representation of the input image. The artificial neurons present in the NN interact, until in the last layer a clear representation of the input is obtained, and a classification is generated by the network. The feature extraction process is depicted in Figure 2.1.

In order to correctly classify a wide range of possible inputs, the Network needs to be trained. The **training process** can be of three main types:

- **Supervised**: Each input is associated to its corresponding label. The network is trained with a kwnowtion of which output is expected.
- **Unsupervised**: Inputs are not labeled. The network has to extract features and classify the inputs independently.
- **Reinforcement Learning**: a reward system is applied in order to modify the network behavior depending on its output.

The training process is responsible for a correct behavior of the NNs, fine-tuning the model until it learns how to correctly classify the inputs.
NNs can be separated into two main classes:

- **Artificial Neural Networks (ANN)**: make use of a mathematical representation of neurons and synaptic connection, loosely inspired by the biology

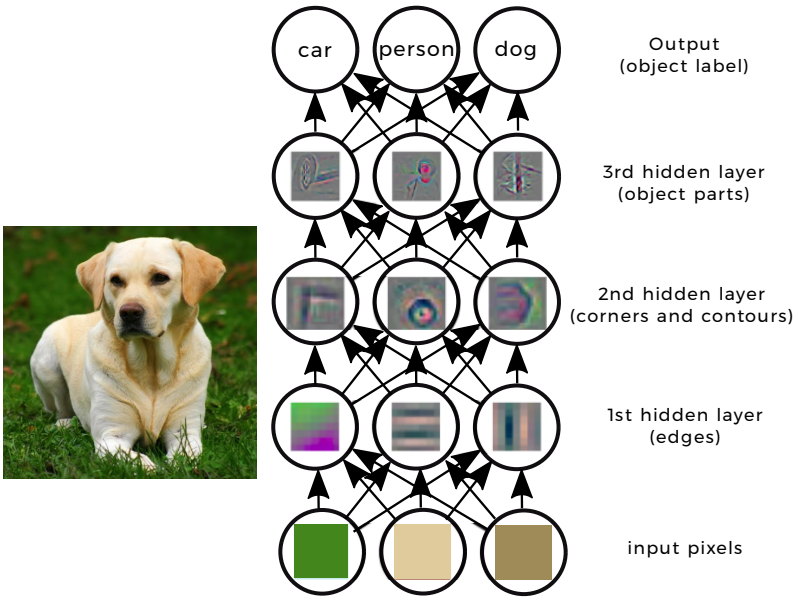


Figure 2.1: Feature extractions in a NN. As we move toward deeper layers, the extracted features get more complex [source: [22]].

of the brain.

- **Spiking Neural Networks (SNN)**: are biologically plausible models based on faithful representation of neurons and synapse interaction.

These two approaches to brain-inspired computing have many differences, which will be presented in the next two sections.
Figure 2.2 contains an overview of the topics discussed so far.

2.3 Artificial Neural Networks

When it comes to solve a classification problem, our brain is a vivid example of a fast and efficient computation: it is capable of executing highly non-linear classification tasks, thanks to its ability to generalize and its neural network hierarchical-based computation.
Taking inspiration from the biological structure of a neuron, a mathematical representation of its structure and behavior is designed and it is called **perceptron**. The perceptron mimicks the behavior of a neuron: it consists of a series of synapses, that allow communication with other neurons, and an output axon which delivers the output computation to the fan-out neurons.
The structure of the perceptron is represented in Figure 2.3. The perceptron receives its inputs from the output of other perceptrons. Inputs are then multiplied

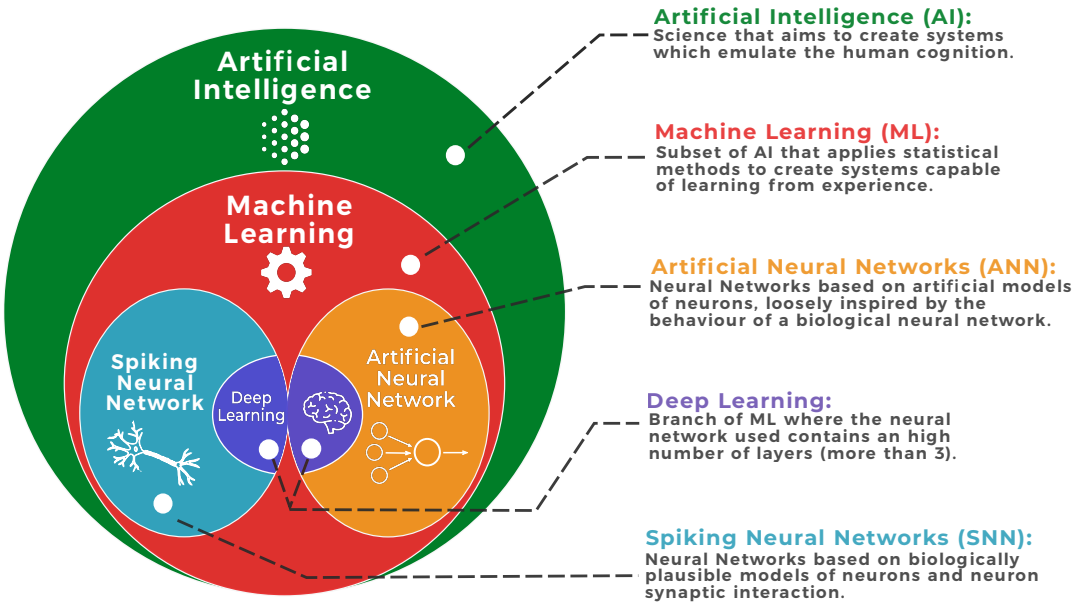


Figure 2.2: overview of artificial intelligence

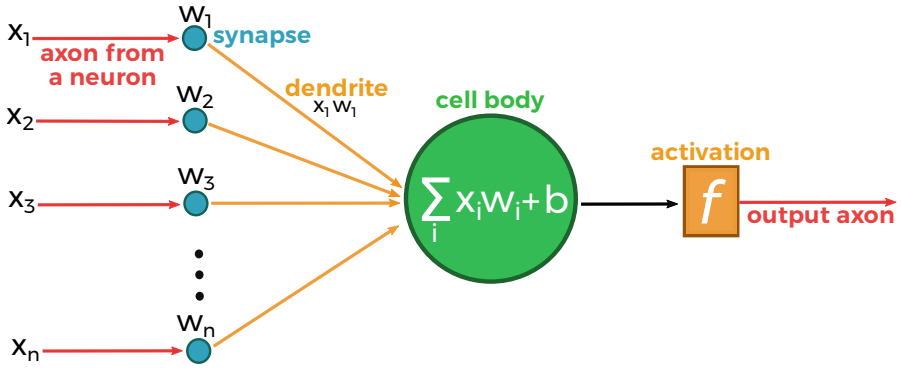


Figure 2.3: Perceptron model

by the synaptic weights, that represent the strength of the connections. In the cell body the weighted inputs are summed in a linear combination, and are added to a bias contribution. An activation function is then applied to the result of the linear combination, providing the output value that is then delivered to following neurons through the output axon.
Usually perceptrons are connected to form large networks, that can be used to learn and apply highly non-linear classification functions to the input data.
The activation functions can be of different kinds:

- **Tanh**: has a characteristic S shape, that saturates the output when it reaches

a high value. The equation is:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- **Rectified Linear Unit (ReLU)**: is defined as $f(x) = max(0, x)$. Allows to get outputs that increase linearly, without a saturation effect.
- **Softmax**: execute a normalization of the output values of all neurons of a layer. It is of use in the last layer, in order to provide a normalized array of outputs whose sum is equal to 1. Its equation is:

$$f(\vec{x}) = \frac{e_i^x}{\sum_{j=1}^{J} e_j^x}, i = 1, 2..J$$

2.3.1 Training process

In order to train an ANN, it is necessary to use a **dataset**. Datasets are sets of examples on which the ANN is trained, and that we expect to be able to correctly classify at the end of the training process.

Datasets

Datasets can be a group of images, or texts or sounds. Data are usually divided into two groups: The *train set*, which contains data that are going to be used to actually train the ANN, and *test set*, that contains data that are not shown to the ANN in the training phase, but are used to evaluate the trained ANN in the inference phase.
When it comes to image recognition tasks, two of the most commonly used datasets are:

- MNIST [8]: A collection of 28x28 grayscale images of hand-written numbers. The dataset comprise a total of 60.000 images for training and 10.000 test images. The total number of classes is 10, that are the numbers 0 to 9.
- CIFAR10 [9]: in this case, images belongs to 10 different classes: Airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck. the image size is 32x32 pixels. Moreover each pixel contains 3 channels, one for each RGB color. Also in this case training set and test set are splitted in 60.000 images and 10.000 images respectively.

All the images contained in a dataset are labelled, in order to know to which class they belong to. Examples of MNIST and CIFAR10 images are reported in Figure 2.4.

Figure 2.4: Example images of MNIST and CIFAR10 dataset

Gradient descend algorithm

A quantitative indicator is needed to compare the output labels predicted by the ANN and the correct labels of the input images. For this purpose, a *loss function* is generally used. Such function measures the distance between the output labels predicted by the ANN and the real labels. The training process is used to gradually modify the weights and biases of the network, in order to minimize the loss function, in such a way that the network predictions are correct for all the input images. In order to achieve such minimization of the loss function, the **gradient descent algorithm** can be used. This algorithm allows to reduce the loss function by computing the derivative of the loss with respect to the weights of the network synapses. The gradient descend requires a series of steps, that are repeated cyclically until the Loss function reaches its minimum. The steps are as follow:

1. First, a forward pass is performed: in this phase, the ANN receives the inputs and compute the resulting predicted labels.

2. The ANN output labels are compared with the correct labels of each processed images. This require the computation of the loss function. Different kind of loss functions can be used, like *Mean Square Error (MSE)*, or *Cross Entropy*, or *Categorical Cross Entropy*.

3. At this point, we need to compute the derivative of the loss function with respect to each weight of the Network. In order to do so, we can apply the

Chain Rule, starting from the weights of the output neurons and then moving backward layer by layer. This stage is called **Backpropagation** [22].

4. Once all the derivative of the Loss function with respect to the weights are computed, we can now modify the weights. This is done by incrementing (or decrementing) their value by a quantity proportional to the computed derivative.

These steps are repeated until the accuracy achieved by the network on the test set reaches a satisfactory value.
The gradient descend algorithm is usually applied on a limited amount of inputs per cycle. This reduces the computational cost of the training, still allowing to reach good training results. The group of inputs used per each cycle is called *batch*: usually the batch size is of power of 2, in order to optimize the hardware utilization. Once all the images belonging to the train set have been used to train the network, an epoch is concluded. The number of epochs needed to train the network depends on the architecture of the network itself.
A key point for a correct training is that the trained network is able to generalize. In fact, a relevant problem in deep learning is the network *overfitting*. This happens every time the network reaches an high accuracy on the images of the train set, but then the accuracy achieved on the test set is far lower. This happens because the network learns to recognize specific features of the train images, but it is not able to generalize those features, and therefore its ability to classify drops once the input image is slightly different from what it is expected. In order to avoid overfitting, some precautions can be applied:

- **Dropout** [23]: This technique consists in turning off some neurons during the training process. When these neurons are off, they are not used by the network and the weights of the relative synapses are not updated. This technique increases the noise during the training process, reducing the co-adaptation of the network to the inputs. When it comes to the inference phase, all neurons and their relative connections are restored. An example of dropout is reported in Figure 2.5.

- **Regularization**: This technique penalize large weights, using penalties during the weight update. This helps to avoid that some weights reaches very high values, leading to overfitting. An example of regularization if the L2 regularization [24]: the weights are update not only by the derivative of the loss by the weight, but also by subtracting a quantity proportional to the square value of the weight itself. This penalizes the presence of large weights in the network.

A network that is capable of high generalization will provide consistent classification results when used on a set of inputs that have not been used in the training

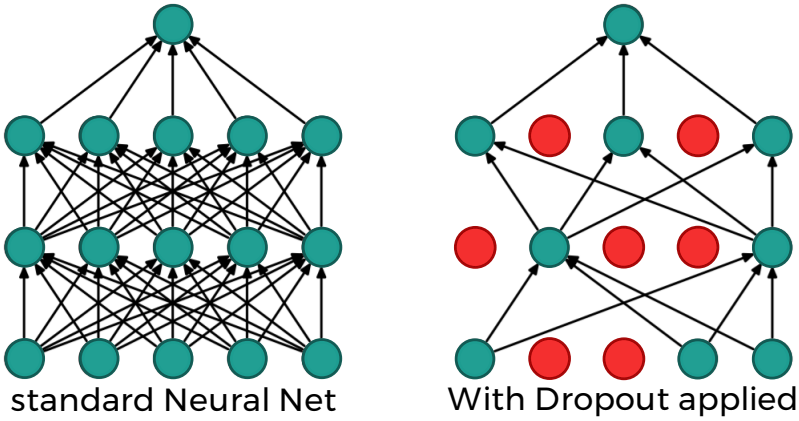


Figure 2.5: Dropout technique [Source: [23]].

process, i.e. never encountered by the network before. On the other hand, a bad classification result during this validation process highlight a bad training process, that is the *overfitting*.

In order to improve the training process, different techniques can be applied. A first solution is to apply an **Optimizer**, which helps to achieve a good learning rate and a fast convergence. Examples of optimizers are *Adam* [25] and *Stochastic Gradient Descent* (SGD) [26]. Also, we can apply **Learning rate decay** [27], that consists in modifying the learning rate used in the training process as the number of training epochs increase.
Finally, **data augmentation** [28] techniques can be used to apply slight modifications to the input images in order to expand the dataset. Some of these image alterations are rotations, flip along the horizontal or vertical axis, and random crops.

2.3.2 Convolutional Neural Networks

In a classical, fully connected NN, all neurons of one layer are connected to all the neurons of the following layer. This comes at a high cost in terms of computation, and power consumption. This kind of networks can not have many layers, because of the high cost that would come in having all the connections required between neurons. Moreover, the NN is more prone to overfitting.
Therefore, some considerations can be made in order to reduce the number of used connections and increase the generalization capabilities of the network.
Images are usually characterized by strong correlations between pixels. The correlation can be of two kinds: **temporal** and **spatial** correlation. For example, let's suppose that we want to recognize a face: we expect to find two eyes, one nose,

and a mouth. Moreover, we also expect that the pixels that contain an eye are neighbour, and cover only a small part of the whole image. Therefore, instead of connecting each neuron to all the pixels of the input image, we connect the neuron to a subset of the input pixels. In this way, we expect to exploit both the spatial and temporal relation among the image pixels. Moreover, this comes with a drastic reduction of the total number of connections needed.
In order to obtain this kind of architecture we use **Convolutional Neural Networks** (CNN) [22]. This networks use convolutional filters, that are applied between two consecutive layers setting the connections among neurons. These filters are applied on a subwindow of the input image, and then are moved progressively in both x and y directions in order to cover the whole image. This allows to recognize a certain feature, for example and eye, independently from its position. As you move the filter on the image, you get a different output depending on the subwindow over which the filter is applied, providing each postsynaptic neuron with a different input quantity.
Each filter is characterize by a *Kernel* value, that is the size of the subwindow that is covered by the filter, and a *Stride* value, indicating the number of pixel shifts done by the filter as it moves in the x and y directions.
An example of convolution is reported in Figure 2.6.

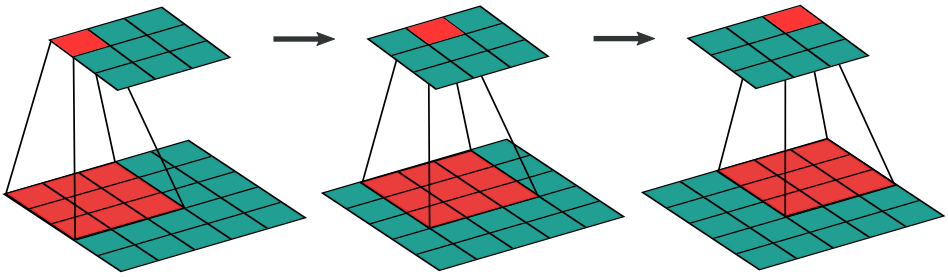


Figure 2.6: example of a filter with a 3x3 kernel, moving with stride = 1 [Source: [29]].

The filter has the same number of channels of the input feature map: for each channel, a different set of weights is learned. The filter is applied multiple times on each channel, moving with a stride value until it covers the total image size in both the dimensions. Every time the filter is applied, each kernel weight is multiplied by the intensity of the corresponding pixel of the input feature map. The output pixel is the sum of the output value obtained by applying the kernel on all the input channels. To this quantity, it is usually added a bias contribution. In figure 2.7, a numerical example of a convolution operation is presented.

On each layer of the NN, a different set of filters are applied. The number of filters that are used on a layer defines the number of channels of the feature map of the following layer. As we move forward into the NN, the number of filters

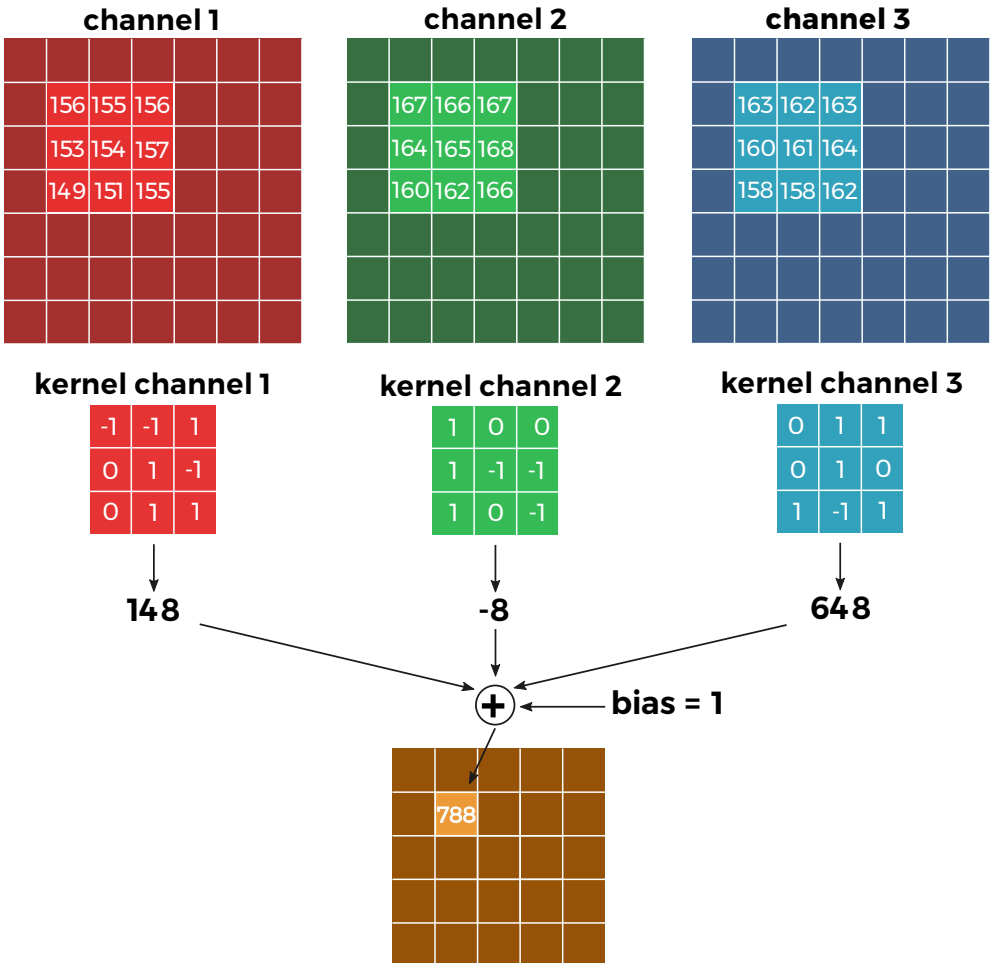


Figure 2.7: Numerical example of a convolutional filter application. For each channel, a different set of weights are learned. For each input subwindow, an output value is obtained by multiplying the kernel weights with the intensities of the input pixels. Finally, the results are summed up adding also a bias contribution. The final quantity is the input of the following neuron, on which the activation function will be applied [Source: [29]].

applied generally increases, and the size of the feature maps decreases. When a filter is trained, it will be able to recognize specific features of the input image: the more we move toward the last layers of the network, the more complex become the features recognized by the filters. Figure 2.8 shows an example of a CNN with 3 convolutional layers and 1 fully connected layer.

In addition to the convolutional layer, in a CNN we can also find **Pooling** layers. This filters are not used to learn new informations, but are generally

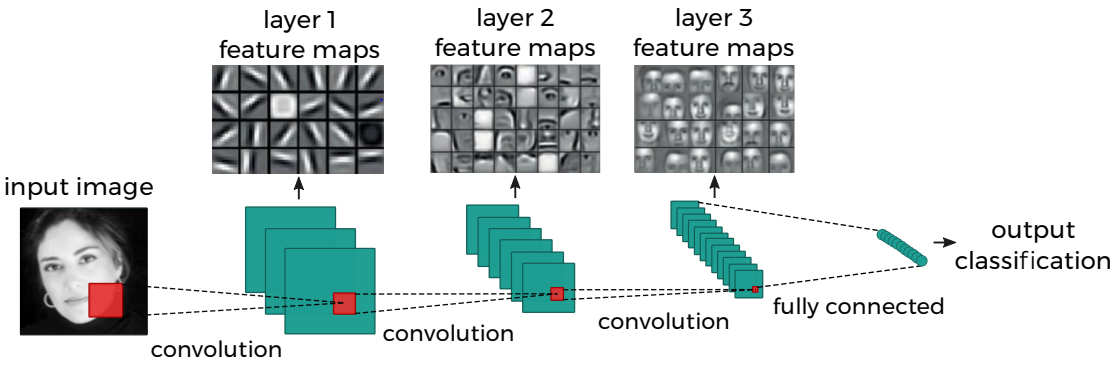


Figure 2.8: example of a CNN with 3 convolutional layers [Source: [30]]

applied in order to get a size reduction of the feature map. Each pooling filter is applied independently on each channel of the feature map, and are characterized by their size and stride values. They move over the input feature map just like a convolutional filter, applying the pooling operation on all the subwindows. Pooling layers can be of two kinds:

- **MaxPooling**: the output is the maximum value between the pixel intensities of the input feature map.

- **AveragePooling**: The output is the mean value between the pixel intensities of the input feature map.

2.4 Spiking Neural Networks

Spiking Neural Networks (SNNs) [19] are a novel yet promising approach in the world of deep learning. Usually referred to as the third generation of NNs, they are based on biologically plausible models of neurons [31], which communicate asynchronously through series of spikes. Neuroscience has provided more and more precise models of the biological neuron that describe with a high precision how the information is processed and delivered through a net of neurons.
When it comes to the implementation of a spiking neuron, different models can be taken as a starting point. Models can be more or less faithful to the biological world, and the one to adopt really depends on the application. As the model complexity increases, it will provide more precise and accurate results, requiring a higher computational cost for its implementation.
The most employed neuron model is the Integrate and Fire (IF) model, which will be described in the following section.

2.4.1 Integrate and Fire (IF) model

The IF is the most widely used neuron model, thanks to its relatively simple implementation. The model is based on a RC circuit, as shown in Figure 2.9. The model can be described considering two neurons, called presynaptic and

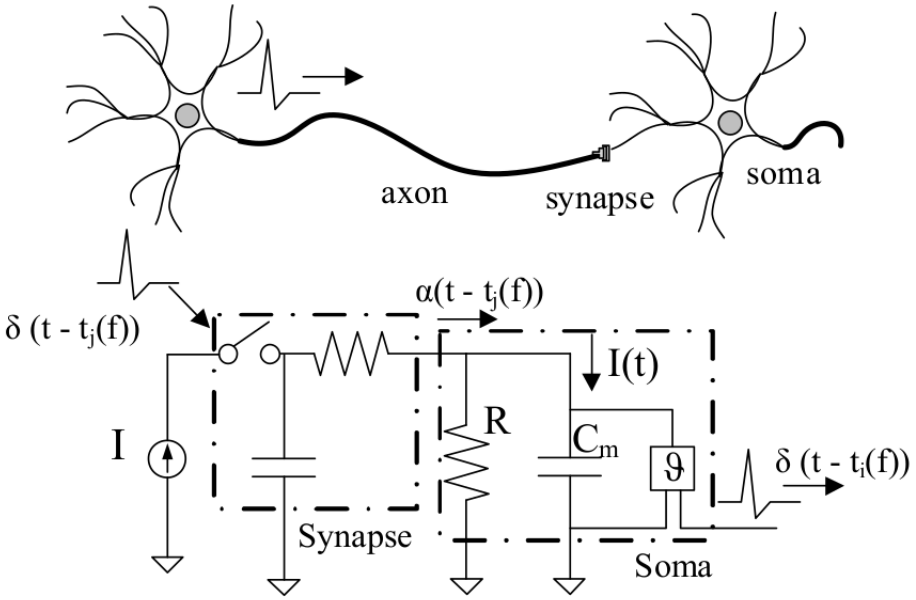


Figure 2.9: Top: biological model of neurons. Bottom: electrical circuit of the integrate and fire model. [Source: [32]]

postsynaptic. The presynaptic neuron connects its axon to the dentrites of the postsynaptic neuron, and the connections are called synapses.
To analize the model, let's consider the case in which the presynaptic neuron emits a spike at time $t_j^{(f)}$ [33]. This spike (or *action potential*) travels through the axon and gets to the synapse. Looking to the RC model, the spike $\delta(t - t_j^{(f)})$ is low-pass filtered at the synapse and a pulse current $\alpha(t - t_j^{(f)})$ is generated. The differential equation that describes the evolution of the postsynaptic membrane potential (PSP) is:

$$I(t) = \frac{v(t)}{R} + C\frac{dv}{dt}$$

from which we derive the membrane time constant τ_m:

$$\tau_m \frac{dv}{dt} = -v(t) + RI(t)$$

When the membrane potential reaches a certain threshold θ at time $t_i^{(f)}$, the postsynaptic neuron produces a spike $\delta(t - t_i^{(f)})$. After the spike, the membrane potential

is reset to a value v_{rest} that is always lower than θ. For $t > t^{(f)}$, the membrane potential starts increasing again as new spikes are received at the input.
A modified version of the IF model, called **Leaky-integrate and fire** (LIF) [33], takes also in consideration the concept of *refractory period*. After the spike, the membrane potential is unable to increase for a short perioid of time Δ^{abs}. During this time, the voltage does not increase even if a train of spikes is received at the input.

The evolution of the membrane potential over time is illustrated in Figure 2.10. When the neuron receives multiple input spiketrains from a multitude of presynaptic

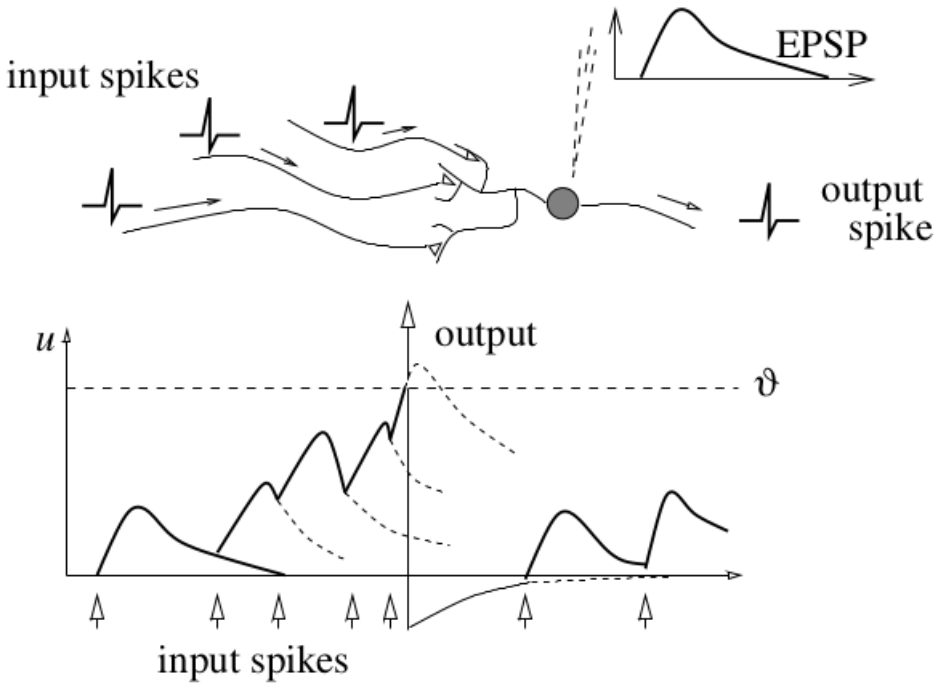


Figure 2.10: membrane potential dynamic [Source: [33]]

neurons, the membrane potential dynamic over time can be described with the following equation:

$$\tau_m \frac{dv}{dt} = -v(t) + (i_0(t) + \sum w_j i_j(t))$$

In this case, it has been made clear the contribution of each spike to the final potential. When the spike is received, a synaptic current $i_j(t)$ is generated and its magnitude is modulated by the synaptic weight w_j. Finally, a constant contribution called bias current $i_o(t)$ is added.

2.4.2 SNNs architecture and advantages over ANNs

The structure and behavior of an SNN are presented in Figure 2.11.

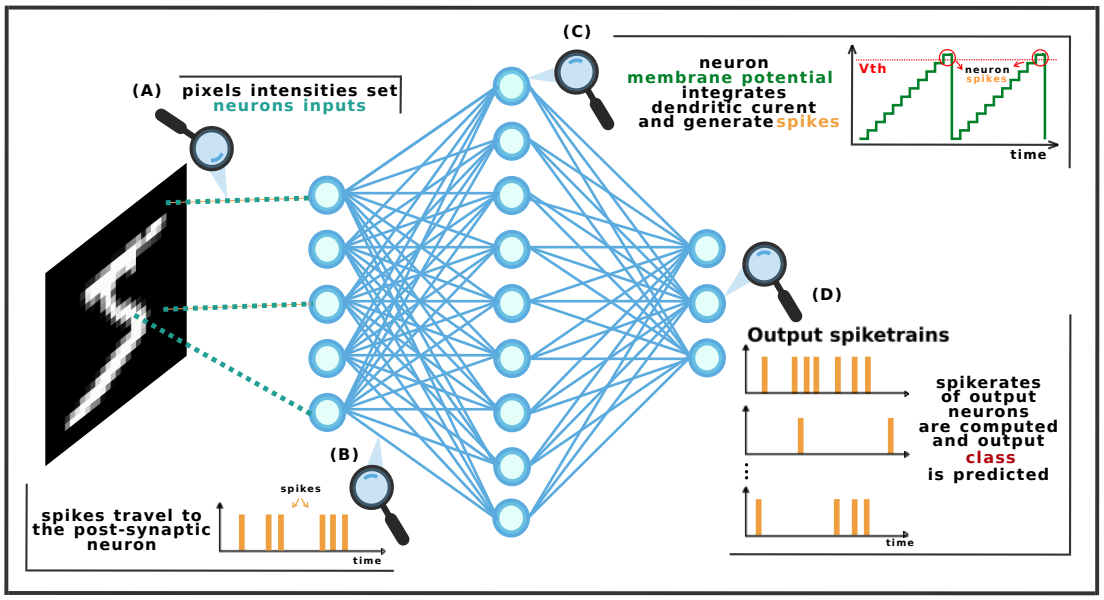


Figure 2.11: Structure and main steps for an SNN for image classification task. In this example, a fully connected SNN, with neurons represented as circles and synapses as lines, is shown. **(A)** Pixel intensities set the inputs of neurons in the first layer. **(B)** The spikes from the pre-synaptic neuron travel across the axon and accumulate in the dendritic tree of post-synaptic neurons. The membrane current of the post-synaptic neuron integrates the incoming weighted spike trains. **(C)** The neuron membrane potential integrates the bias current and the membrane current. An output spike is generated each time the potential reaches a predefined threshold. Afterwards, the membrane potential is set back to the initial level. **(D)** The output neurons, one for each class, generate spike trains: for each neuron, its corresponding spike rate in a predefined time-window is computed, and used as the output prediction for its class.

SNNs major improvements over traditional ANNs are the following [31]:

- The intrinsic asynchronous, spike-based communication protocol adopted in the network allows to *reduce the power consumption* required in the computations. In fact, in a classical ANN the power consumption is independent from the intensity of the network inputs, because the network computations are executed no matter of the incoming inputs. On the other hand, when little or no spikes are received at the input the SNN does not compute, thus consuming not much power. At the same time, when sudden burst of input spikes are received, the neurons of the

SNN elaborate the information and power consumption increases. Such intrinsic low power behavior originate from the assumption that the information received from the outside is sparse, bringing to a drastic power-efficient computation.

- The asynchronous, spike-based design makes this networks ideal to *cooperate with event-based sensors*. The events provided as input can be seen as train of spikes directly processable by the network.
- The asynchronous data-driven computation brings to a fast propagation of information through the network. This can lead to a *pseudo-simultaneous* processing of information, in the sense that a first approximation of the output classification is available immediately after a first spike is received at the input. This is also true when the number of hidden layers is high, because spikes propagates through layers of the network as soon as a sufficient amount of spikes is delivered by the lower layers neurons to the following layers.

Information representation through spikes

In order to provide input spikes and collect the resulting output spikes of the SNN, we have to understand how to properly code continuous informations using spikes. Different approaches are used to obtain such conversion [34]:

- *Rate coding*: In this case, the information is coded as the mean firing rate of generated spikes in a defined observation period.
- *Inter-spike interval (ISI)*: the intensity of the activation is coded as the precise delay between consecutive spikes.
- *Time to first spike (TTFS)*: In this code technique, the information is encoded in the latency that goes from the beginning of the stimulus, to the time of the first output spike. This solution enables a very fast information processing, carrying enough information.

The three coding techniques are presented in Figure 2.12.

SNN Training

When it comes to SNNs training, different solutions can be applied depending on the typology of training that we are looking for: unsupervised or supervised.

When we want to train our network with an **unsupervised** technique, a possibility is represented by the *Spike Time Dependent Plasticity* (**STDP**) [35]. This solution is based on the biologically plausible synaptic weight modification that occur as trains of spikes are processed by locally connected neurons. The basic

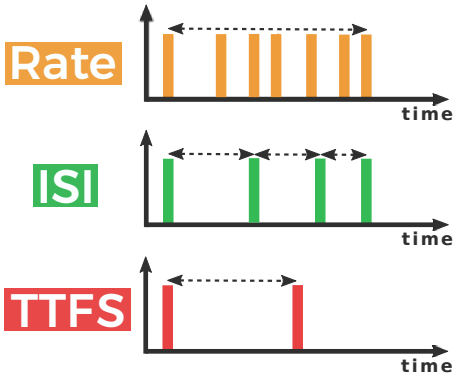


Figure 2.12: Temporal diagram of the presented coding techniques

idea is that we expect to get the output neuron to spike when a certain pattern of spikes is detected at the input of the network. The learning rule is reported in Equation 2.1 [35].

$$w_i \leftarrow w_i + \Delta w_i,\ \Delta w_i = \begin{cases} +a^+ w_i(1-w_i), & \text{if } t_{out} > t_i \\ -a^- w_i(1-w_i), & \text{if } t_{out} \leq t_i \end{cases} \tag{2.1}$$

t_i and t_{out} refer to the spike time of the presynaptic and postsynaptic respectively. The synaptic weight w_i is incremented by a quantity Δw_i if the presynaptic spike arrives before the output postsynaptic spike. On the other hand, if the postsynaptic spike is generated earlier than the input spike, the weight w_i is reduced. The amount by which the weight is reduced is proportional to the weight itself and to the learning rates a^+ and a^-.

For what concerns **supervised** learning, a fundamental problem arises: classical supervised learning approach, i.e., backpropagation, cannot be applied due to the non-differentiability of the SNN loss function [36]. Therefore, two main approaches have been proposed to achieve supervised learning in SNNs:

- Use the backpropagation algorithm directly in the spiking domain. This method generally requires to substitute the loss function with a placeholder function, that can be differentiated [37][38].
- Train an equivalent ANN model in the ANN domain and then convert the trained network to the SNN domain [7].

Throughout this research, we focus on the latter approach. Training the network in the ANN domain allows us to use the current state-of-the-art training policies and techniques. Moreover, the ANN-to-SNN conversion has shown promising results, allowing to get SNNs that reach the same, or very close levels of accuracy,

compared to their corresponding ANN versions [36][7]. However, some precautions and limitations have to be considered when using this approach, as we will explain in our analysis in Chapter 3.

2.5 Intel Loihi Neuromorphic Chip

ANNs achieve the best results in terms of accuracy and efficiency when executed on highly parallel hardware like GPUs (Graphics Processing Unit), and even more with specialized hardware accelerators, like Google TPU (Tensor Processing Unit) [18]. Similarly, SNNs require their specialized hardware to achieve the best results in terms of power efficiency and latency [39].
Neuromorphic chips represent an efficient hardware solution when it comes to the implementation of SNNs. The highly parallel asynchronous structure, combined with the hardware implementation of the neuron model, such as the *leaky-integrate-and-fire* model [35], allows to achieve far better results both in latency and power efficiency with SNNs when compared to their CPU and GPU implementation. Recent developments in the field of neuromorphic hardware have brought valid and powerful solutions for the simulation of spiking neural models, like IBM TrueNorth [3], SpiNNaker [4] and Intel Loihi [5]. Our research has focused on the latter platform.

The **Intel Loihi** is a neuromorphic processor providing highly parallel and power efficient asynchronous computation. The chip is based on a neuromorphic mesh of 128 neurocores which execute the neuron computations. The management of all the neurocores is possible thanks to 3 embedded x86 processors. Finally, an asynchronous network-on-chip (NoC) connects neurocores allowing neuron-to-neuron communication [5].

2.5.1 Neuron model

The biologically-plausible neuron model adopted by the Loihi architecture is based on a modified version of the *Current-Based (CUBA) leaky-integrate-and-fire* model [5].
Each neuron is characterized by its synaptic response current $u_i(t)$ and its membrane potential $v_i(t)$. Given a postsynaptic neuron i, it receives in input a train of spikes that are sent by a presynaptic neuron j. These spikes can be represented as a train of Dirac delta functions:

$$\sigma_j(t) = \sum_k \delta(t - t_k)$$

where t_k is the spike time.
When the train of spikes arrives at the synapse, it is filtered by a synaptic filter input response $\alpha_u(t)$, which is defined as:

$$\alpha_u(t) = \frac{e^{-\frac{t}{\tau_u}}}{\tau_u} H(t)$$

Where H(t) is the step function. Each filtered spike train is multiplied by the synaptic weight w_{ij} associated to the synapse that connects neurons n_i and n_j. The synaptic response current can then be computed as the sum of all the weighted and filtered spike trains, with an additional bias current b_i:

$$u_i(t) = \sum_j w_{ij}(\alpha_u * \sigma_j)(t) + b_i$$

Finally, the synaptic current is integrated by the membrane potential $v_i(t)$.

$$\dot{v}_i(t) = -\frac{1}{\tau_v} v_i(t) + u_i(t) - \theta_i \sigma_i(t)$$

When the membrane potential reaches a certain threshold θ_i, the neuron spikes. After that, the membrane potential is reset to a v_{rest} value and start increasing again as new input spikes are received. The time constant τ_v is responsible for the *leaky* behaviour of the model [5].

2.5.2 Chip Architecture

Overview

Each neurocore can simulate up to 1024 spiking neural compartments units: each compartment can emulate a tree of neurons. Neurons variables are updated at every algorithmic time-step. The spikes generated by a neuron are delivered to all the compartments belonging to its synaptic fan-out through the **Network on Chip** (NoC). The NoC allows to deliver spikes between different neurocores in a packet-messaged form, following a mesh operation that is executed over a series of algorithmic time-steps. In the absence of a global clock, a barrier synchronization mechanism is used to ensure that at the end of each time-step, all neurons are ready to proceed to the next time-step. An off-chip communication interface allows to extend the mesh up to 4096 on-chip cores, and up to 16,384 hierarchically connected cores [5].
The architecture of a single Loihi chip is displayed in Figure 2.13.

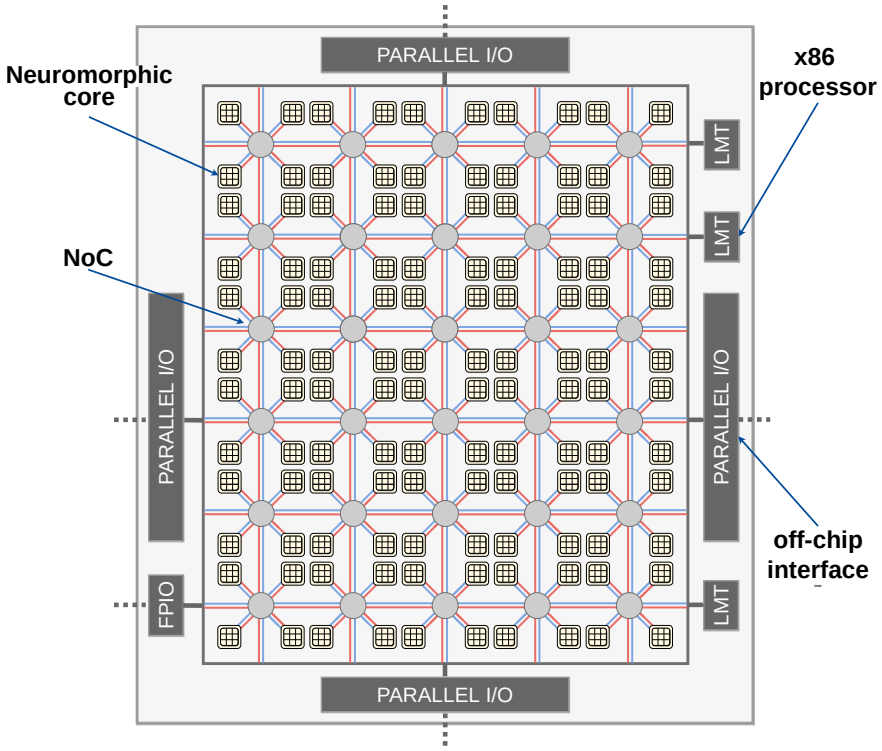


Figure 2.13: Loihi single chip architecture

Mesh operation

The Network on Chip allows the neurons to exchange spikes during each algorithmic timestep t. Each neuron inside the mesh neurocores goes through the different stages of its computation, generating output spikes that has to be delivered to all the postsynaptic neurons that belong to the neuron fan-out [5]. In Figure 2.14(B) we have an example of two neurocores, A and B, containing two spiking neurons n_1 and n_2: thes spikes are delivered to the destination neurocores through the NoC. In order to correctly deliver the spikes, each neurocore iterates over its neurons, and deliver the spikes to the neurocores that contain the postsynaptic neuron. This operation is performed through the NoC, that is based on a dual-channel router network. Spikes are sended by the neurocores on one of the two channels, which allows the trasmission of 1 spike message at a time. Each neurocore processes a list of fan-out cores in order to deliver more than one output spike.
The asynchronous digital architecture of the Loihi chip requires a barrier synchronization mechanism [5] that allows all the neurons present in the neural network to advance to the next algorithm timestep $t + 1$. When a neurocore ends the distribution of spikes of its neurons, it send a barrier synchronization message that

is received by the neighbour neurocores. At this point, all spikes that are still traveling are flushed. After this, a second message is shared to all cores and as soon as it is received by a neurocore, it advances to the next timestep $t + 1$.
An example of the mesh operation is presented in Figure 2.14

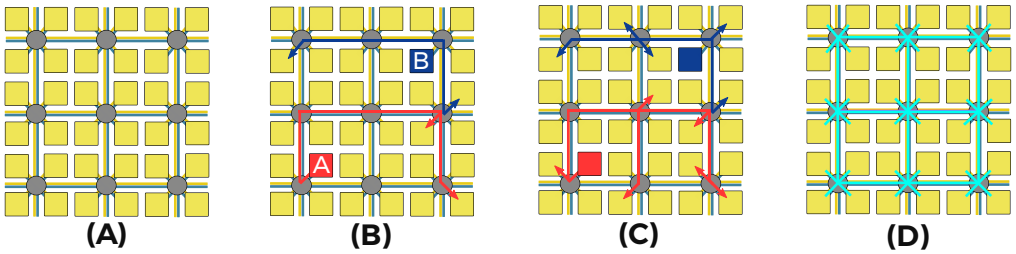


Figure 2.14: Loihi neuromorphic mesh operations: **(A)** neurons advance through their computations. **(B)** Neuron n_1 and n_2 belonging to neurocores A and B generate output spikes that are delivered to postsynaptic neurons belonging to other neurocores. **(C)** other neurons spike and the spikes are sended thorugh the mesh. **(D)** the barrier synchronization mechanism is exchanged and neurons move to the following timestep [Source: [5]]

Moving to a closer examination of the neurocore message exchange protocol, let's consider the case of a neuron x which generates spikes to be delivered to a group of postsynaptic neurons $A - F$, as presented in Figure 2.15. Each postsynaptic neuron may belong to different neurocores, thus a core-to-core connection protocol has to be established [5].
Each destination core is connected to the sender core with an axon connection, labeled with a specific ID. The axons are exclusive for each core-to-core pair, and connect neurocore's output and input edges.
Some constraints has to respected when it comes to the neurocore mapping of neurons and their input and output connections. These limitations are reported in Table 2.1.

Neurocore constraints	
max compartments	1024
max fan-in axons	4096
max fan-out axons	4096

Table 2.1: Constraints of Loihi neurocores [5].

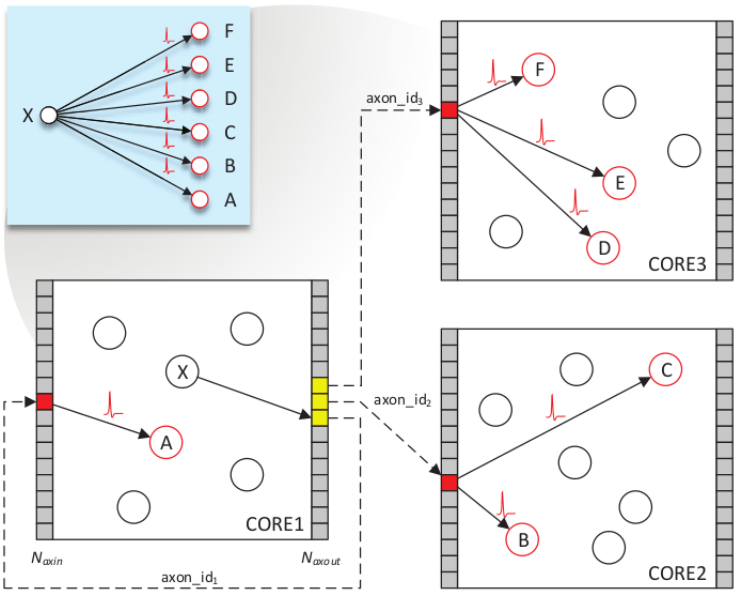


Figure 2.15: Loihi neurocores connection policy. In yellow are the core output edges. In red are the core input edges. [Source: [5]]

Neurocore microarchitecture

Figure 2.16 shows the schematic of the Loihi neurocore microarchitecture. Besides the input and output blocks, the design can be divided into 4 units [5]:

- The **Synapse** unit processes the incoming spiketrains, and retrieves the corresponding synaptic weights from memory.
- The **Dendrite** unit modifies the synaptic current u and membrane potential v of each neuron present in the neurocore (up to 1024 neurons).
- For each neuron, the output messages are create by the **Axon** unit. Each message is associated to the specific postsynaptic neuron that has to be delivered to.
- Finally, the **Learning** unit modifies the synaptic weights of all the synapses connected to each neuron. The synaptic weights are updated accordingly to the defined learning rule.

2.6 Simulators for SNNs

When it comes to the software implementation of SNNs, different simulators can be employed. Usually they allow to describe the network with a high-level of

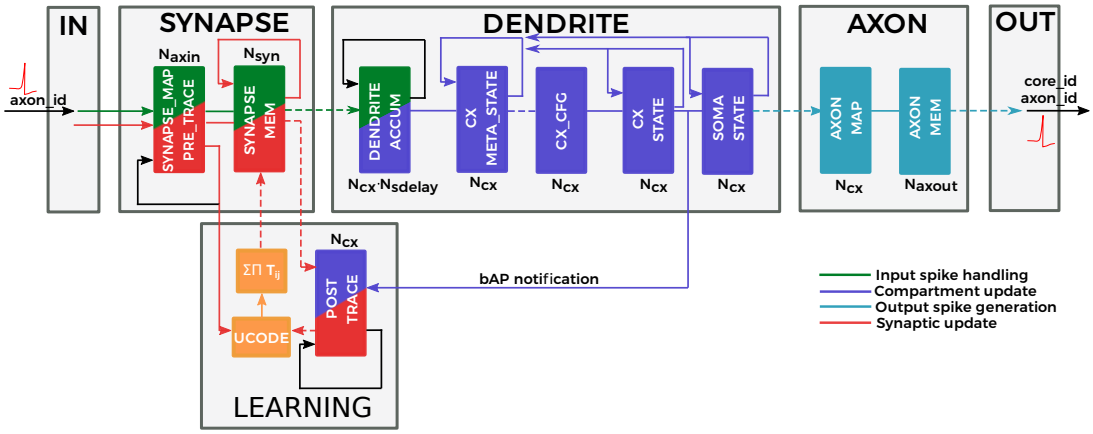


Figure 2.16: Loihi neurocore microarchitecture. The color codes refear to the core operation modes. **Green**: input spike handling. **Purple**: compartements update. **Cyan**: spike generation. **Red**: synaptic weight update. [Source: [5]]

abstraction, giving to the user the possibility to model large ensembles of neurons and build large networks. However, it is also possible to fine-tune the characteristics of each neuron and synapse present in the network, leading to a deep control over the whole simulation.
Different simulators are available with such a scope: *Nengo* [40], *BRIAN2* [41], *PyNN* [42] are all software toolkit written in python that let to implement large networks of neurons with a high level of detail in the neuron model description. These simulators can run locally and can be used to execute simulations of complex SNNs.
Specifically related to Loihi neuromorphic chip, a dedicated software toolkit called *Intel NxSDK* [43] is available. This python API allows to build network models with deep controls of neurons characteristics, and directly deploy the model on Loihi hardware. The software API gives the user a complete control over the neuromorphic chip, allowing a full exploitation of the hardware.

Concerning the ANN to SNN conversion, specific software tools need to be employed. Some simulators have their own conversion toolkit, like *Nengo-DL* [44] for the *Nengo* simulator. However, a more universally compatible conversion toolkit is **SNN-ToolBox** [7]. This software takes as input ANNs that have been trained with different backends, like Keras [45], and convert them into SNNs models compatible with different SNN simulators, like *PyNN*, *BRIAN2* and a built-in simulator called *INI* [7]. Moreover, SNN-ToolBox offers a full compatibility with *Intel NxSDK*, allowing to build converted SNNs that can be directly deployed on the Loihi platform.

An in-depth analysis of SNN-ToolBox will be reported in section 3.4.2.

2.7 Event-Based cameras

Event-based cameras are bio-inspired sensors for the acquisition of visual information [6]. This new technology is based on an asynchronous paradigm for dynamic light acquisition. In a classical frame-based image sensor, the image recording of a scene is obtained by stacking a series of frames collected with a defined temporal rate, that is called *frame rate*. This approach lead to the collection of information that is temporally not related to the dynamic of the scene.
On the other hand, in a event-based sensor, the information recorded is directly related to the light variations in the scene. The camera works asynchronously, not recording frames with a precise timing. Instead, the sensor records negative and positive brightness variations in the scene. Thus, each pixel encodes for a brightness change in the scene. Pixels are independent, and can record both positive and negative light variations.
Figure 2.17 shows a comparison between frame-based and event-based recording.

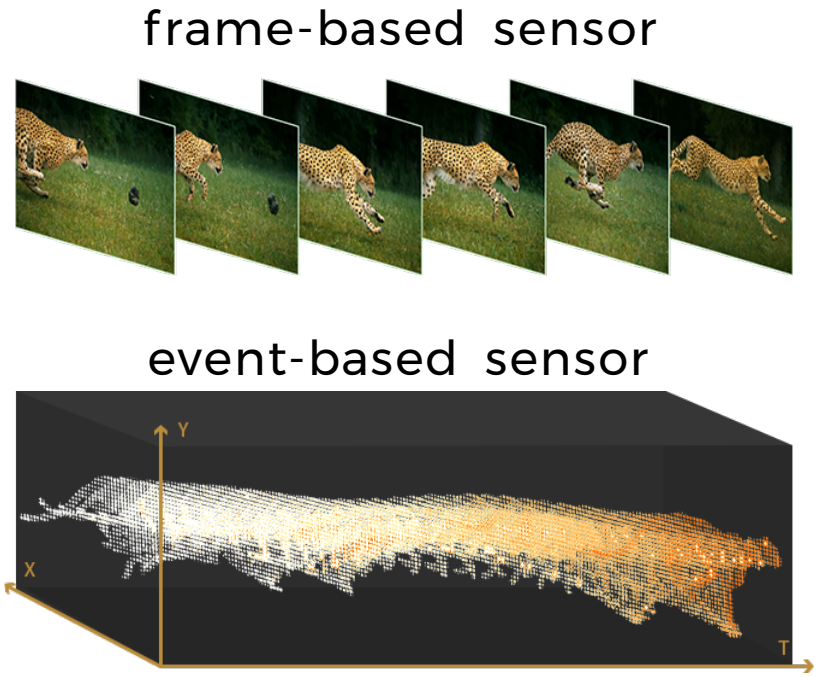


Figure 2.17: Comparison of frame-based and event-based recordings. In the first case, frames are recorded with a precise timing. In the event-based case, the only information collected is the movement of the cheetah through time, because is the only dynamic subject in the scene [Source: [46]].

The output stream recorded by the sensor is a sequence of *events*, each coding for a specific brightness change in the scene at a specific position (x,y),
This completely new data acquisition mechanism brings to consistent benefits [6]:

- *Power consumption*: The event-based recording paradigm lead to a consistent power reduction. In fact, new data is recorded only when a bright variation is detected in the scene. This means that, in the absence of light changes, no information is recorded, leading to a almost zero power consumption.
- *High dynamic range*: The sensor has a dynamic range of 140 dB, that is far higher than frame-based camera (60dB). This allows to use the sensor also in very low light conditions.
- *High temporal resolution*: the sensor can record light variations with a very high temporal resolution of the order of microseconds.

Figure 2.18(A) shows an example of a DVS camera, the DAVIS 240. In Figure 2.18(B) a typical event-based output is reported.

Figure 2.18: **(A)** example of a DVS camera, the DAVIS 240. **(B)** typical output of an event-based camera: Magenta pixels denote positive events, whereas cyan pixels code for negative events [Source: [47]].

Chapter 3

ANN-to-SNN Conversion

3.1 Theory of conversion

3.1.1 Introduction

The ANN-to-SNN conversion represents a valid solution when it comes to the training of large, convolutional SNNs. The possibility to train the ANN with state-of-the-art training methodologies, combined with the highly power efficient and low-latency design of spiking networks, can set a new standard in the field of Deep Learning. The fundamental idea behind the conversion process is that there is a close link between ANN neurons output activation function (ReLU) and the output spiking frequency of SNN neurons. Essentially, it is possible to define an equivalence between the output activation of classical artificial neurons and the spikerates of biologically inspired neurons.

After the training, for each connection among two neurons of consecutive layers *i* and *i+1*, the weight $w_{i,i+1}$ is defined. Moreover, for each neuron of layer *i+1*, also the bias b_{i+1} is derived. In the equivalent SNN model, these parameters need to be translated into an equivalent value for the spiking neural model. Specifically referring to the *CUBA LIF* neuron model implemented by Loihi and described in section 2.5.1, the conversion works as follows:

- the bias b_{i+1} is associated to the bias current b_i of the neuron n_{i+1}.
- The weight $w_{i,i+1}$ is directly set as the weight of the synapse connecting neurons n_i and n_{i+1}.

Besides the learned parameters, each layer of the ANN has to be converted to an equivalent spiking version. This means that each layer will be composed of equivalent spiking neurons that follow the *CUBA LIF* model.
In order to get a proper conversion, the trained parameters of the ANN must

be efficiently converted into the corresponding parameters of the SNN: this also requires to take into consideration the intrinsic differences between the two models, and some adjustments are consequently required to get a correct conversion. In the following sections we will present the main steps that have to be followed during the conversion process, along with the parameters that influence the final result.

3.1.2 Activation vs Spikerate

As previously mentioned, the basic principle behind the conversion process is the assumption that the output activation of an analog neuron can be seen as the mean spikerate of an equivalent spiking neuron. This equivalence is particularly true in the case of the ReLU activation function, because of its linearity, such that the activation increases directly with its input. The same behavior can be noticed for the spikerate of a biological neuron, where the spikerate increases as the number of input spike increases. However, there might be some inconsistencies between activations and spikerates after the model conversion. Different explanations are possible for potentially poor results [7]:

- the input spikerates are insufficient to modify the membrane potential in such a way that the membrane potential crosses its threshold. This may lead to a lower output spikerate with respect to the artificial activation.
- The number of spikes produced by the spiking neuron are too high, leading to a higher spikerate with respect to the artificial neuron activation.

Increasing the simulation time can solve the first problem, giving to the spiking neuron more time to generate output spikes. On the other hand, a reduction of the number of simulation timesteps may lead to solving the second problem, reducing the amount of spikes generated by the neuron in the unit of time. However, none of the previous solutions lead to a correctly converted network, where the neuron spikerate is equivalent to the corresponding artificial activation.

3.1.3 Normalization process

The solution usually adopted to solve the inconsistencies between artificial activations and spikerates is the **normalization** process. The amount of output spikes produced by a neuron is strictly related to the values of synaptic weights and memebrane potential. If the weights are too high, the probability for the membrane potential to cross the threshold increases, leading to an excessively high amount of output spikes. Therefore, the normalization process adjust the weights to have output spikerates that are consistent with the artificial activation. As we modify the weights, we are consequently varying the memebrane threshold to input weights

ratio.
The normalization can be of two types [7]:

- Conservative approach: The first possibility consists of rescaling all the synaptic weights of a layer by the maximum positive artificial activation of that layer. This solution ensures that the activations of all neurons of a layer are ≤ 1. This can be done by scaling both weights and biases: if we call λ_l the highest activation of layer l ($max[a_l]$), then the weights and biases of layer l are scaled as:

$$W_l = W_l \frac{\lambda_{l-1}}{\lambda_l},\ b_l = b_l/\lambda_l$$

  When this normalization is applied, we can be sure that no spikerate saturations will be present in the final SNN layer. However, this solution may lead to very low firing rates across the network, being the whole process fine-tuned for the worst case scenario (highest activation). In fact, especially in the case of strong positive outliers in the layer activations distribution, the firing rates of all the other neurons will be far lower than expected.

- Percentile approach: to avoid an excessive reduction of weights due to strong outliers, we can choose as λ_l the $p-th$ percentile of the total activity distribution of layer l. This solution highly reduces the problem related to extreme outliers in the distribution. In this case we will experience spikerates saturation for those neurons that have activations higher than the $p-th$ percentile. However, we reduce the probability of having very low spikerates in the layer neurons. Moreover, by choosing a $p-th$ percentile between [99.0, 99.9], we ensure that the spikerate saturation regards very few neurons.

3.1.4 Input conversion

When it comes to the application of an input image to a converted SNN, there are 2 possibilities that we can go for.

1. A first solution is to convert the analog value image into a spiking equivalent image. This require to adopt some kind of analog to spike conversion, like a Poisson spikerate generation [48]. However, this solution is usually avoided in the case of converted SNN. In fact, using a train of spikes as input for the input neurons produces an inconsistency among the ANN and SNN input format: ANN input neurons are trained to deal with constant quantities, the pixel intensities. On the other hand, the input neurons of the spiking network receive train of spikes that are generated by a Poisson model, leading to impairs in the performance.

2. A second solution consists in associating the pixel intensity to the bias current of the corresponding input neuron. This results in a constant input to the neuron synaptic current, which consequently lead to a steady increase of the membrane potential. Thus, the spikerates of the input neurons are consistent with their equivalent artificial activations.

3.1.5 Pooling layers

As explained in 2.3.2, two possible pooling layers can be adopted in Analog networks: MaxPooling and AveragePooling. The conversion of an AveragePooling layer can be executed without problems, simply setting the synaptic weights to $1/n$, where n is the number of pixels of the feature map sub-window considered by the filter. On the contrary, the MaxPooling implementation in the spiking domain can be more complex: a possible solution can be to use a winner-take-all algorithm on the ouput spiketrains on which the pooling is applied. However, this can be hard to implement and generally the MaxPooling is converted into an AveragePooling layer during the conversion [7].
To avoid the possible conversion loss due to the change in the layer adopted, it is recomended to use AveragePooling layers directly in the ANN domain.

3.2 Adapting the conversion to Loihi hardware limitations

The conversion process requires a series of additional considerations that have to be taken into account to get a successful conversion when the SNN is specifically built for the Loihi platform.
First of all, the Loihi architecture uses **limited precision** (8bit) synaptic weights, that can be defined within the interval [-256,255] with steps of 2 or within the intervals [0,255] or [-128,127] with steps of 1. On the other hand, the trained ANN uses **full precision** 32bit floating point weights. Therefore, a preliminary quantization of the ANN-trained weights is crucial to get a precise converted SNN. In this quantization process, the distribution of the input weights has a major role in the outcome of the conversion: the input weights has to be clipped into the Loihi quantized range, therefore a tight weight distribution can be mapped to the quantized interval without relevant errors. On the other hand, the presence of outliers in the original weights distribution can be the main source of an inprecise conversion: this is due to the fact that a large amount of weights are clipped to fit into a reduced number of quantization steps. This leads to possible inconsistencies between the pre and post quantization weight distributions. Figure 3.1 shows the quantization problem described so far.

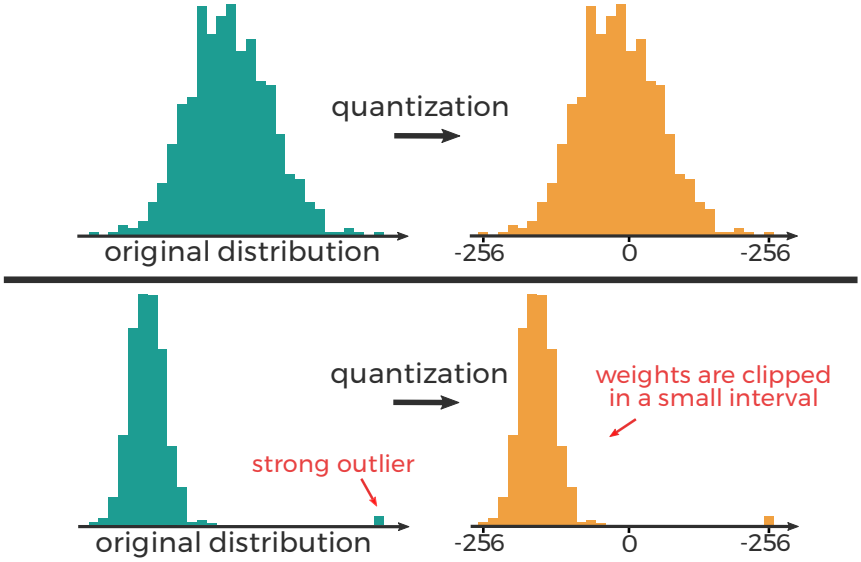


Figure 3.1: examples of weight quantizations. In the first case, the weight distribution does not contain strong outliers, leading to a seamless quantization. In the second case, the original weight distribution presents a very high value, that we refear to as a strong outlier. Given that all weights have to be quantized in the interval [-256,255], a large amount of weights is quantized in a reduce number of steps.

To decrease the probability to find strong outliers in the final trained weights, the **L2 regularization** [24], applied both on activations and kernels during training, helps to keep weights into a limited range, avoiding the precense of very high weights that may affect the conversion.
A good practice to evaluate the quality of the conversion is to look at the **correlation plots** between the ANN layer activations and its corresponding SNN layer output spike rates. Figure 3.2 shows three typical correlation plots that can be obtained with good and bad conversion processes. The expected behaviour is having all the points along the diagonal $y = x$: this means that each artificial activation is properly translated into the spikerate output of the corresponding neuron.

Plot 3.2(A) is an example of a good correlation: as can be seen, the ANN activations are properly converted into SNN spikerates, being all the points distributed along the main diagonal. On the contrary, plot 3.2(B) shows a worse conversion: the ANN activations and the SNN spikerates are still distributed along the diagonal, but the distribution of points is not confined to the desired range, being spread all over the plot. Finally, plot 3.2(C) is another example of bad conversion. However,

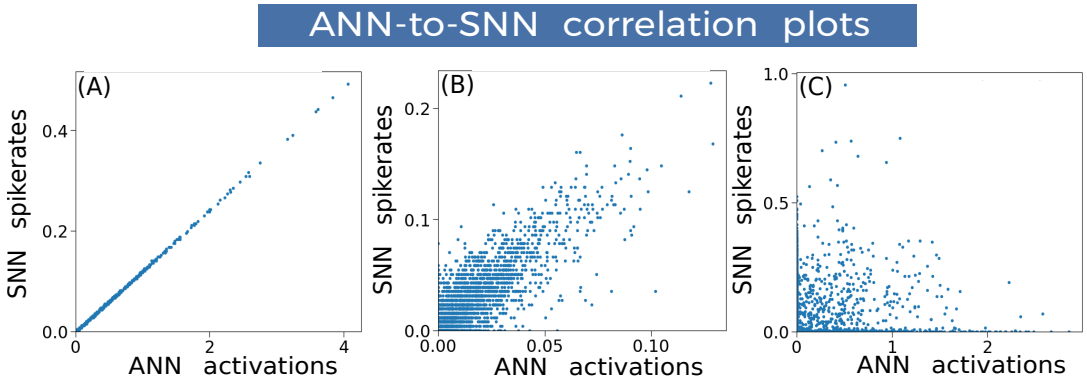


Figure 3.2: Examples of correlation plots.

in this case, the activations and the spikerates are totally unrelated.

3.3 Tunable Conversion Parameters

Many parameters can be tuned during the ANN-to-SNN conversion process, and a detailed analysis over their effects on the converted SNN is necessary. These parameters modify the spiking neuron model, the characteristics of the network and the simulation duration.

- **Reset mode**: The reset mode defines the behavior of the neuron after a spike. As previously said, the neuron spikes every time its membrane voltage exceeds the threshold V_{th}. After the spike, the membrane voltage is reset to a value that depends on the chosen reset mode:
  - *Hard Reset*: The membrane voltage is reset to a fixed reset value V_0 every time the neuron spikes. This solution is less computationally expensive, but less accurate.
  - *Soft Reset*: The membrane voltage is reset to a value equal to the difference between the highest value reached by the membrane voltage and the membrane threshold. This solution is more accurate, but more expensive, because the amount of compartments needed to simulate each neuron is doubled, compared to adopting the hard reset.

  The two reset mode are shown in Figure 3.3.

- **Desired Threshold to Input Ratio** (**DThIR**): As described in Section 3.1.3, the weights of the input ANN model has to be converted to synaptic weights of the SNN. Because of the limited dynamic range of spiking neurons, the output of a spiking neuron may saturate due to an excessively high input, given by some

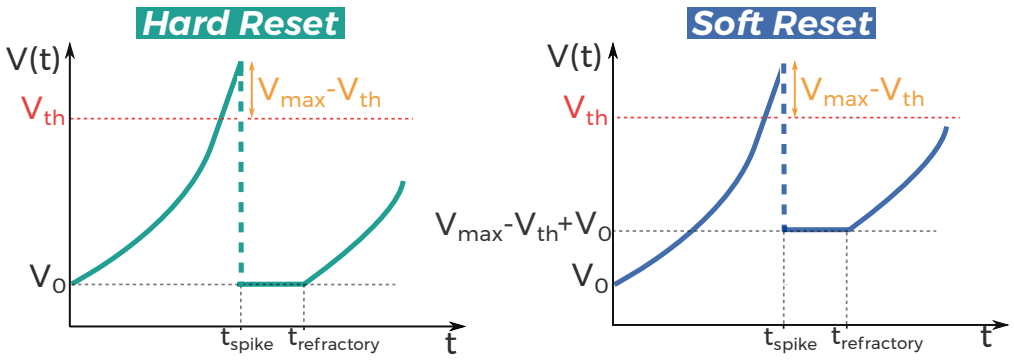


Figure 3.3: Examples of hard reset and soft reset

out-of-scale synaptic weights. Hence, it is necessary to normalize the network and set a constant ratio between the incoming neuron inputs and its membrane threshold [49].

- **Simulation duration**: This parameter defines the number of algorithmic timesteps for which the network receives the same image as input, i.e. the inference time. A longer duration gives the network more time to output its prediction, but it increases the latency of the system.

3.4 Experimental Setup

The experimental setup that has been used to study the conversion process comprise 3 main steps:

- *ANN training*: first, it is needed to properly train the neural network in the original analog domain. The training has been performed using the Python **Keras** API [45], which allows to train the network with state-of-the-art techniques and optimizations.

- *ANN-to-SNN Conversion*: once the network has been trained in the analog domain, it is possible to convert it to a spiking equivalent model. The conversion is performed using **SNN-ToolBox** [7], an open-source python conversion tool compatible with Loihi.

- *SNN inference on Loihi* The Loihi platform can be programmed using Loihi **NxSDK** [43], a python API that allows to create and test spiking models on Loihi hardware.

The described steps are depicted in Figure 3.4. In the following 3 sections, we are going to describe the tools used during the conversion process.

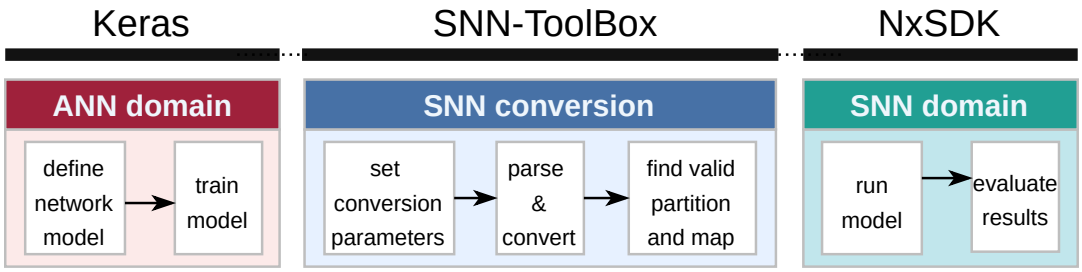


Figure 3.4: Tool flow of our simulation process.

3.4.1 ANN training

To study the behavior of the conversion, a network has been used to evaluate the process. Such a network, that we will refer to as *NxNet*, is a convolutional neural network that contains only convolutional layers and a final dense layer. Its structure is reported in Table 3.1.

Layer	features	Kernel	stride	size MNIST	size CIFAR10	Activation
Input	1			28x28x1	32x32x3	ReLU
Conv2D	16	4x4	2	13x13x16	15x15x16	ReLU
Conv2D	32	3x3	1	11x11x32	13x13x32	ReLU
Conv2D	64	3x3	2	5x5x64	6x6x64	ReLU
Conv2D	10	4x4	1	2x2x10	3x3x10	ReLU
Flatten				40	90	
Dense				10	10	SoftMax

Table 3.1: NxNet architecture for MNIST dataset.

To achieve a better conversion, both activation and weight **L2Reguralization** are applied on the network layers: in both cases the regularization parameter is set to $1 \cdot 10^{-4}$. The use of regularization during training is preferable for preventing the divergence of the parameter distribution and avoiding information loss due to the quantization process of the parameters, as discussed in Section 3.2.

The datasets on which the analysis have been performed are MNIST [8] and CIFAR10 [9] datasets. For each input image, the intensity values are normalized between 0 and 1. Both networks are written in Keras, using TensorFlow [50] as backend. The training is performed with the following policies:

- learning rate decay: initially set to 0.001, it is halved after 15 consecutive epochs without validation accuracy improvements, with a minimum value reachable of $5 \cdot 10^{-7}$.
- Adam optimizer [25].

- Small data augmentations, with width and height shifts of 0.1, and 10° rotations.

After training, the values of test accuracy achieved by the networks are reported in table 3.2.

Nework	Dataset	Accuracy
NxNet	MNIST	98.79%
NxNet	CIFAR10	78.92%

Table 3.2: Accuracy results of the ANN models.

3.4.2 SNN-ToolBox

To apply the ANN-to-SNN conversion, we use *SNN-ToolBox* (SNN-TB) [7], an open-source conversion tool that is compatible with Loihi's Python NxSDK [43], that will be described in Section 3.4.3. More specifically, SNN-ToolBox executes the conversion and provide a spiking model that is compliant with NxSDK specification. The results obtained with the conversion process may not always be optimal, due to some limitations of the NxSDK API and specific constraints of Loihi neurocores. Therefore, in the following Section 3.5, we present a case of study for an ANN-to-SNN conversion, specifying a set of general guidelines to follow in order to achieve a converted SNN that reaches the same accuracy levels of the corresponding ANN.

Conversion Process

The SNN-ToolBox conversion requires four main steps:

1. **Parsing**: The input ANN model is analyzed, extracting the relevant infromation related to the layer architecture. Then, an equivalent ANN model, called *parsed model*, is built. This new model will contain only those layers that are necessary for inference, discarding layers that are used only during the training process (i.e. Dropout, BatchNormalization). Moreover, if the model contains MaxPooling layers, they are converted into equivalent AveragePooling layers. This is because, as we explained in 3.1.5, AveragePooling is the only supported pooling operation in the spiking domain. The parsed model is the one used as reference for the following conversion.

2. **Conversion**: A NxSDK-compatible spiking model is obtained, applying a normalization process that adapts weights and biases to the limited dynamic of the spiking neurons, satisfying the selected value of *DThIR*.

3. **Partition**: The conversion process requires to find a valid partition of the neural network on the Loihi chip. Some constraints has to be respected in order to

have a valid partition: these constraints, reported in table 2.1, are related to the synaptic fan-in and fan-out of each neurocore, and the maximum number of neurons that can be mapped on a single neurocore.

4. **Mapping**: The partition is mapped onto the Loihi chip, and the model is now ready to be used in the SNN simulation.

3.4.3 Loihi NxSDK

The Intel NxSDK [43] is a sotfware development tool that can be used to control and program Intel Loihi chip. The tool allows to control the chip with two levels of abstractions:

- **NxNet**: an high level API that allows to control the chip without having a deep understanding of the underlying hardware. It is possible to specify our model in terms of groups of neurons, connections and specify the learning rate.

- **NxCore**: this low level API gives the full control of the hardware implementation of neurons. It requires knowledge of neurocore configurations, and per-core, per-register and per-field programming.

Concerning the implementation of SNN obtained after conversion from an original ANN, a NxSDK module called **NxTF** is used. This module allows to build a spiking equivalent model from an original analog network. The ANN training has to be performed using Keras, that is the only NxTF ANN training compatible API. Not all Keras layers are supported by the Loihi's Python NxSDK. The only supported layers are the one in Table 3.3. This limitation has to be taken in consideration in the development of the ANN architecture.

Dense	Flatten	Reshape	Padding
AvgPooling2D	DepthwiseConv2D	Conv1D	Conv2D

Table 3.3: Layers supported by NxTF.

The access to Loihi chip is possible through the Intel NRC server. Five different Loihi device boards are available for the user: each one differs in the number of available Loihi chips. Moreover, boards are accessible for a limited period of time. The available boards are reported in Table 3.4.

	Loihi	Loihi_2h	Nahuku08	Nahuku32	Nahuku32_2h
numer of available chips	4	4	8	32	32
execution time limit	20 min	2 hours	20 min	20 min	2 hours

Table 3.4: Avaible Loihi boards

3.5 Results

The simulations reported are executed on the **Nahuku32_2h** board, which is the best choice both in terms of available Loihi chips (32), and available execution time. As described in section 3.3, the three main parameters that have been analyzed for a fine tuning conversion are *reset mode*, *DThIR* and *simulation duration*. Different simulations have been realized to evaluate the effects of these parameters on the final SNN accuracy of the chosen network.

3.5.1 Results varying the DThIR

In this experiment we evaluate the conversion results varying the DThIR. The simulation duration is set to 256 algorithmic time-steps that, as we will see later on, is a reasonable choice for both soft reset and hard reset. The tested DThIR levels are $2^1, 2^3$ and 2^5. Selecting higher levels is usually not a good solution because the membrane potential threshold may gets too large. The results are reported in Figure 3.5(A).

- MNIST: In both cases of soft reset and hard reset, the SNN accuracy is equal to the ANN accuracy value for DThIR = 2^1 and 2^3. However, when the parameter is increased to 2^5, the accuracy drops in both soft and hard reset cases.
- CIFAR10: Also in this case the highest accuracy is reached for DThIR=2^1, both for hard and soft reset. However, the accuracy starts reducing when the DThIR is set to 2^3, and gets to a minimum when the parameter is increased to 2^5.

As a consequence of these results, a value of DThIR = 2^1 is chosen for the following analysis.

3.5.2 Results varying the duration and reset mode

This analysis tries to find a good compromise between simulation duration and reset mode. Choosing a longer duration, we expect to get more precise results, paying in terms of output latency. Moreover, the use of a soft reset is expected to provide higher accuracy. The results are reported in Figure 3.5(B).

Looking at the results achieved on the MNIST dataset, a test accuracy of 98.70%, only 0.09% lower than the one obtained with the ANN model, is reached in the

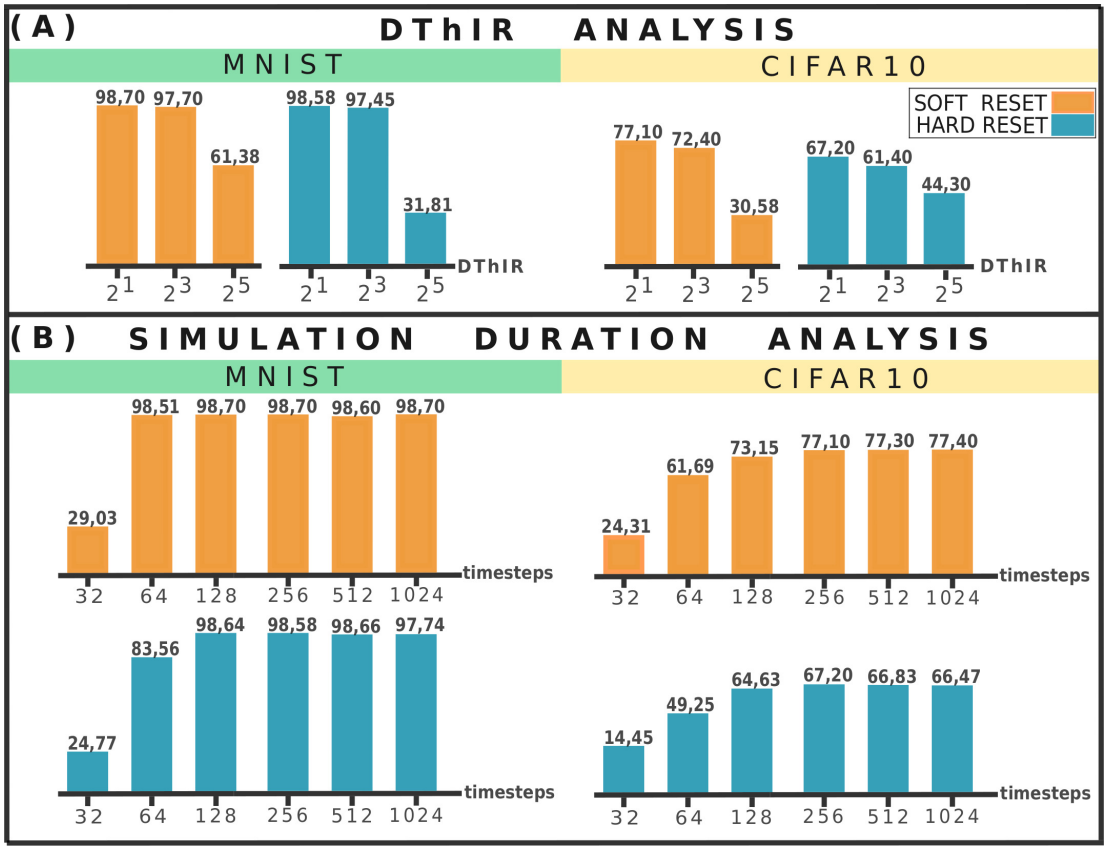


Figure 3.5: The legend is common for all the plots. Classification accuracy results for the NxNet on the MNIST and CIFAR10 datasets. **(A)** Varying the DThIR with fixed simulation duration of 256 timesteps. **(B)**Varying the simulation duration with fixed DThIR = 2.

soft reset case, when the simulation duration is longer than 64 time-steps. On the other hand, it takes at least 128 time-steps for the hard reset case to get to the same level of accuracy. Moreover, the accuracy reached by both soft/hard reset remains stable also in the case of longer simulations.

The results for the CIFAR10 dataset clearly show that in the hard reset case the ANN accuracy of 78.92% is never reached. The maximum value of accuracy is 67.20% when the simulation gets longer than 256 time-steps. On the other hand, the soft reset shows better results than the hard reset, even if it does not achieve the same results as the corresponding ANN. An accuracy of 77.10% is reached with 256 time-steps, slowly growing to 77.40% with a longer simulation of 1024 time-steps.

Taking in consideration the case of a simulation duration of 256 time-steps, we have the following average time for single image classification and chip occupancies

reported in Table 3.5. Looking at the number of occupied neurocores, in both MNIST and CIFAR10 cases the soft reset makes use of more cores.

Reset Mode	Dataset	Classification time	Neurocores
soft	MNIST	8.312 ms	27
hard	MNIST	6.464 ms	20
soft	CIFAR10	21.371 ms	37
hard	CIFAR10	26.159 ms	29

Table 3.5: Accuracy results of the ANN models.

3.5.3 Correlation Plots Analysis

For better understanding the reasons why the soft reset conversion achieves better results than the hard reset conversion, we compare the correlation plots of the converted layers. Figure 3.6 shows the correlation plots of all the convolutional layers, both for the soft reset and the hard reset versions, and on both datasets. In each of the 4 presented cases, a simulation duration of 256 time-steps is applied, as well as a DThIR = 2^1. At a first glance, it is immediately clear that the correlation plots of the soft reset conversion are far more compliant with the expected behavior when compared to the hard reset results, both in the case of MNIST and CIFAR10 datasets. Looking at the MNIST - soft reset simulation, the correlation plot of the first layer shows a perfect conglomeration of activations (x axis) vs. spikerates (y axis) points along the main diagonal. This means that the conversion of the layer is working as desired, having all neurons spiking with a rate equivalent to their corresponding ANN neurons activations. The same principle is adopted for the following layers.
Looking at the MNIST - hard reset simulation, all layers correlation plots show a far worse conversion result with respect to the soft reset case. Starting from the first layer, the points distributes with a overlapped-staircase behavior. The same happens in the second layer, where it is also present a dilatation of the agglomerate of points along the x axis. However, both in the 3^{rd} and 4^{th} layers correlation plots, the points are sufficiently compacted along the diagonal, and in fact the final accuracy achieved by this network is equivalent to the ANN model, as seen in Figure 3.5.
Regarding the CIFAR10 analysis, the soft reset case gives good correlation plots, even if the points form a thicker agglomerate with respect to the MNIST case. On the other hand, the hard reset gives far worse results: the correlation between activations and spikerates is far less evident, with a general behavior that follows the one of MNIST case, but much more emphasized. The analysis of these plots clears the 10% accuracy drop in the hard reset case, as seen in Figure 3.5.

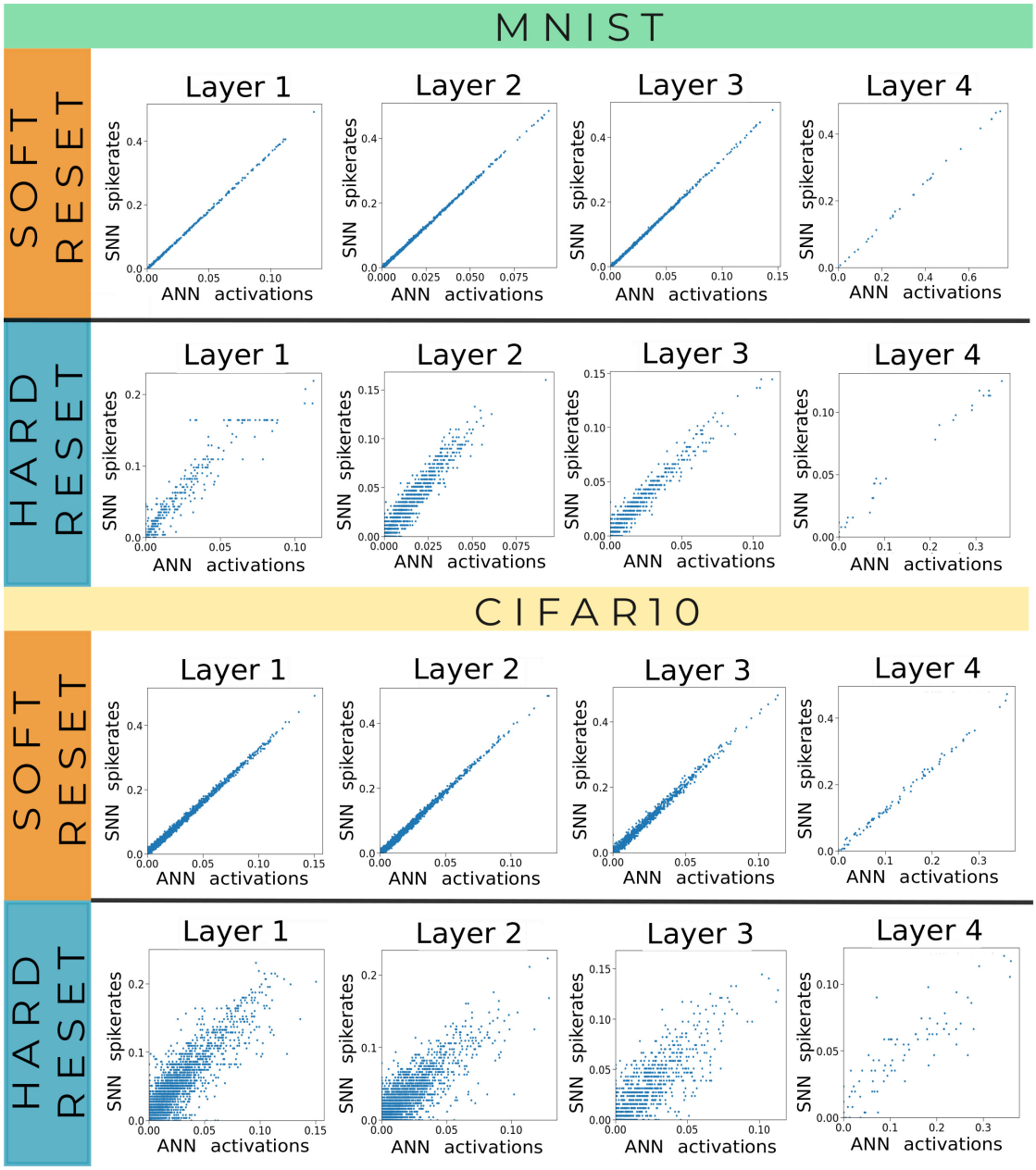


Figure 3.6: Correlation plots of the first 4 layers of NxNet. The first column shows the results on the MNIST dataset, whereas the second column presents the results for the CIFAR10 dataset.

Overall, the results obtained on the CIFAR10 dataset are worse than the one obtained on MNIST, both in the case of soft reset and hard reset. This can be addressed to the higher complexity of CIFAR10 images, representing a much

challenging dataset to work with.

3.6 Conclusions

Overall, the use of a *soft reset* mode gives higher accuracy results, because of the lower information loss that occurs during the conversion, as clearly shown by correlation plots. A good choice for the simulation duration seems to be ≥ 256 time-steps: a faster simulation may lead to an accuracy loss, as shown in the CIFAR10 case. On the other hand, using more than 512 time-steps does not lead to higher level of accuracy, as shown in both MNIST and CIFAR10 analysis. Finally, a DThIR value equal to 2^1 seems to be the best choice to reduce the loss during the conversion.
Furthermore, the conversion results are also strongly influenced by the ANN architecture, as well as by the ANN training policies. To have a deeper evaluation of the conversion process, several other ANN models have been trained and converted. These models vary in terms of size, number of layers, and layers characteristics. Problems generally arise when the ANN layers are too wide, making the conversion not feasible because neurocore constraints are violated. Therefore, when it comes to build very large networks, it is suggested to use *depthwise separable convolutional layers* that require less core occupation than equivalent convolutional layers. A wider analysis concerning ANN design adopting such layers will be discussed in Chapter 5.

Chapter 4

An Efficient Spiking Neural Network for Recognizing Gestures with a DVS Camera

4.1 Introduction

Recent developments in the field of event-based cameras have increased the possibilities to use such devices in combination with neuromorphic hardware to realize low latency, low power systems for image recognition. As explained in section 2.7, the great advantages of these sensors in terms of power efficiency, latency and high dynamic range sets them as a valid alternative to frame-based cameras, especially in the field of embedded systems.
In order to develop reliable SNNs that can work properly in combination with such devices, we need datasets that can serve as test-applications and prove the benefits of this technology.
As a consequence, the IBM event-based gesture dataset [10] has been used in this research to realize a gesture recognition system that use a SNN to provide gesture classification of video recorded with an event-based camera.

4.2 IBM gesture dataset

The IBM DvsGesture dataset [10] is a fully event-based gesture recognition dataset. Each gesture is recorded with a DVS128 camera, providing a total of 1342 samples divided in 122 trials. In each trial, 1 subject execute the 11 different gestures in

sequence. A total of 29 subjects under 3 different light conditions form the whole dataset.
The gestures are: *hand clapping, left hand wave, right hand wave, left arm counter-clockwise, left arm clockwise, right arm counter-clockwise, right arm clockwise, arm roll, air drum, air guitar* and finally *other gestures* class.

Each gesture has an average duration of 6 seconds and is composed by a collection of all the events that have been recorded by the DVS camera. As explained in section 2.7, there are 2 types of events: a positive event is recorded every time a positive variation of light is detected, whereas a negative event is generated when the detected light variation is negative. An example of the described event-based gesture recordings is reported in Figure 4.1.

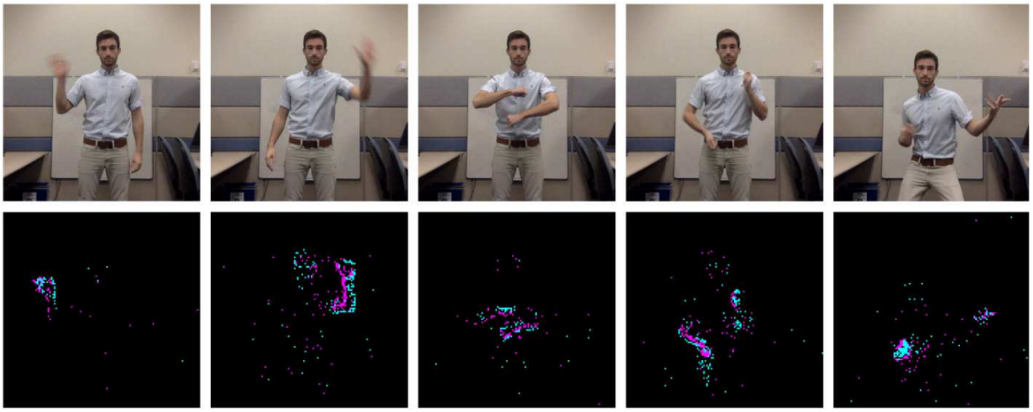


Figure 4.1: Examples of DvsGesture recordings. First row: actual frame of RGB recordings. Second Row: events recorded during the execution of the hand gestures; magenta pixels represent positive events, whereas cyan pixels are generated by negative events. [Source: [10]]

Event-based data are ideal when used as input to SNNs, thanks to their intrinsic asynchronous and spiking behavior. However, in the context of our research, we are training a network in the ANN domain and only in a second stage we convert it into the SNN domain. This force us to find an alternative representation of the input data, being the ANN not trainable on pure sequences of events.
A valid solution can be to train the ANN with a series of frames obtained by collecting the incoming events. However, some choices have to be made to achieve a good conversion into frames:

- Choose the amount of events to collect into a single frame.
- Select the size of the frame and its number of channels.
- Set a policy for positive and negative events accumulation.

4.3 Dataset conversion

4.3.1 Events Accumulation

As reported in [36], there are two accumulation approaches:

- *Time-based accumulation*: all events that occur in a fixed time window are accumulated in a single frame (Figure 4.2(A)). This solution ensures that the timing information within frames is respected.

- *Quantitative-based accumulation*: a fixed number of consecutive events are accumulated in a single frame (Figure 4.2(B)). This solution guarantees that each frame will have the same amount of information. However, this may not be a good choice when it comes to gesture recordings. In fact, the number of events generated by a gesture in a fixed time window also depends on the type of the gesture itself. Not all the gestures generate the same amount of events per second: gestures that involve wider movements generate more events per second.

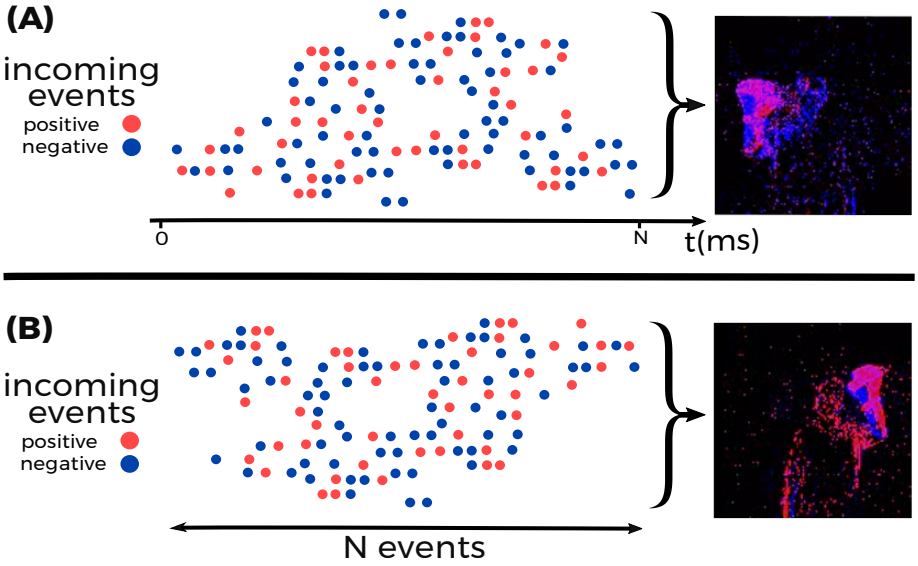


Figure 4.2: **(A)**: example of time-based accumulation during a period of N ms. **(B)**: example of quantitative-based accumulation of N events.

Therefore, using a *quantitative* approach, the number of frames generated depends on the number of events produced by the gesture. Gestures with the same time length may lead to a different amount of frames, having different event rates. As a consequence, the final dataset will be imbalanced, having a diverse amount

of frames per classes, both in the train set and in the test set, as reported in Figure 4.3. In order to balance the dataset, one may reduce the amount of frames

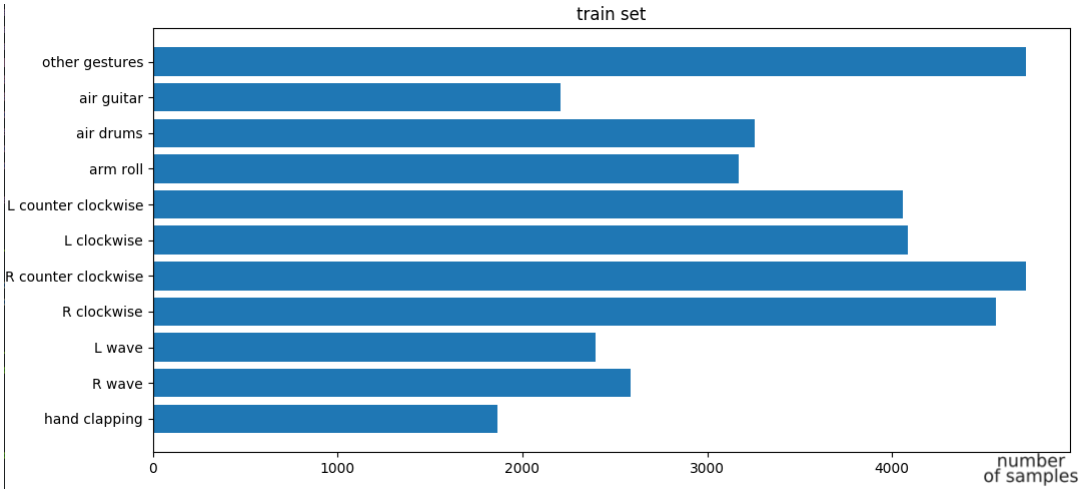


Figure 4.3: Train set sample distribution over classes in quantitative based approach, with an accumulation of 6000 events per frame. As it can be seen, the amount of frame per class differs as the number of events within a fixed time window depends on the motion itself.

per gesture to a number equal for all classes, but this would come out in a drastic reduction of the used information from the original event-based recordings. Hence, based on these considerations, the *time-based* accumulation is preferable, because it guarantees a balanced dataset. Therefore, the results relative to the *quantitative-based* accumulation approach are not discussed.

4.3.2 Time Window Size

The amount of events per seconds varies not only from gesture to gesture, but also between different trials of the same gesture. A mean number of 98 events/ms is estimated by evaluating the original dataset over all the available gestures of all the different trials. This information is a relevant starting point in the choice of the time window size that each frame has to cover. A too short time window may lead to frames that do not contain a sufficient amount of information for a proper classification. On the other hand, a too long time window would make the system unable to work in a real-time condition. In this research, four different time windows are explored: 60ms, 150ms, 235ms and 300ms. Preliminar experiments have found that choosing a time window of less than 60ms would bring to an insufficient amount of events collected per frame. On the other hand, an accumulation time of

more than 300ms would lead to a total of less than 3 frames per second, that we consider as the minimum for a real-time application.

A single frame may also have more than one channel, each covering a subset of the whole time window. For example, a frame covering a window of 300ms can have 3 channels, each covering 3 sub-windows of 100ms. This solution allows to get frames in which the temporal information is preserved, since the channels cover consecutive time sections. An example of such channel temporal separation is shown in Figure 4.4.

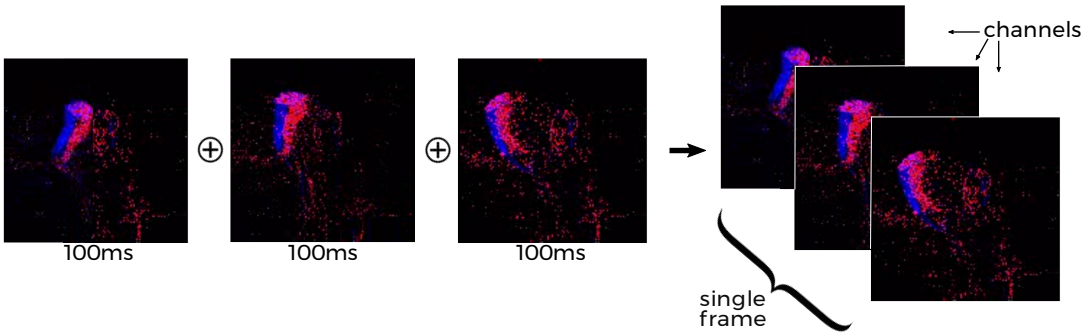


Figure 4.4: Examples of temporal channel separation within a single frame.

Moreover, another solution may be to use overlapped frames, i.e., the time windows covered by two consecutive frames are partially overlapped. For example, using an overlap factor of 2 with frames of 300ms, the frames will cover partially overlapped ranges. The first frame will be $[0ms; 300ms]$ and the following frame will cover the range $[150ms; 450ms]$. A visual example of overlapped frame policy is reported in Figure 4.5. There are several advantages in choosing this solution:

- The number of frames generated from the original dataset is multiplied by the overlap factor, leading to a bigger dataset that guarantees better training results.
- The frames can cover different time windows, augmenting the temporal information in the dataset.
- The throughput of the system is multiplied by the overlap factor.

In our analysis, an overlap factor of 2 has been chosen. Using an overlap factor $n > 2$ would lead to generating redundant overlapped frames. On the other hand, a value $n < 2$ would reduce the benefits of having overlapped frames.

4.3.3 Dealing with Polarity

Each event carries three features: the x and y position of the detected event, as well as the polarity of the event that can be either positive or negative. We need

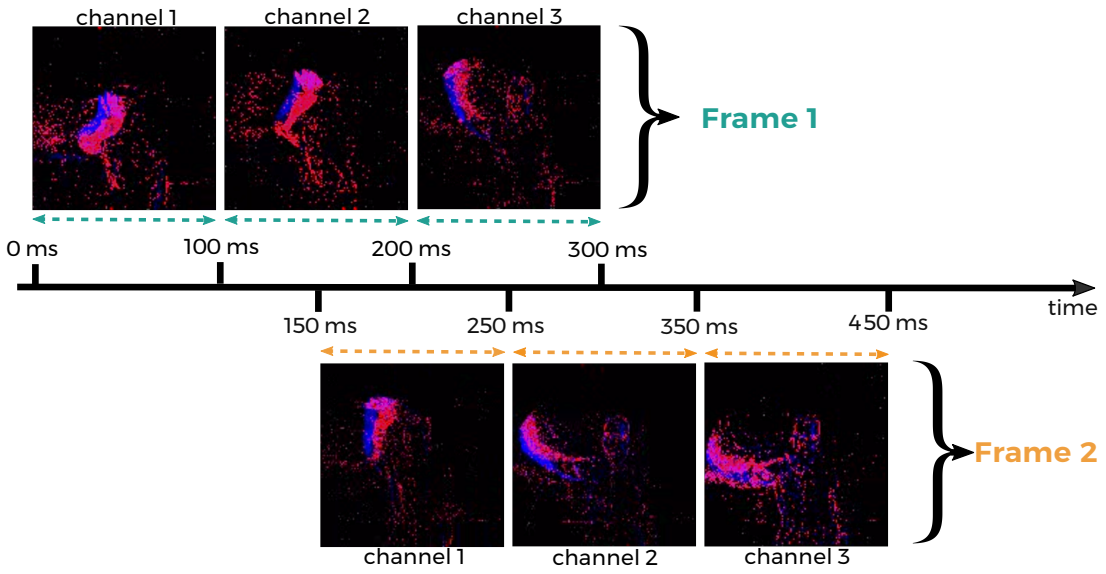


Figure 4.5: Examples of frame temporal overlapping with an overlapping factor of 2.

to establish a policy that defines how to efficiently exploit the polarity information. We can define 3 possible choices:

1. **2 Channel Accumulation** (Figure 4.6(A)): accumulate the positive and negative events in two different channels of the frame, c_+ and c_-. Both the channel pixels are initialized at 0, and when a positive event is detected, the pixel (x, y, c_+) is incremented by 1. On the other hand, a negative event increases the pixel (x, y, c_-) by 1. Finally, pixel intensities are normalized in the range $[0; 255]$. This solution prevents the information loss. In fact, the polarity information may become relevant when the gestures differ only for their sense of rotation, because the accumulation of opposite signed events form a trace of the gesture motion over time.

2. **1 Channel Accumulation** (Figure 4.6(B)) accumulate all negative and positive events on the same channel, keeping the polarity information. All the pixels are initialized to a mean value of 128, and incremented or decremented by 1, depending on the polarity of the event. Finally, pixel intensities are normalized in the range $[0; 255]$.

3. **Discard Polarity** (Figure 4.6(C)): discard the polarity information and collect all the events in a single channel, simply incrementing the pixel (x, y) every time either a positive or a negative event occurs. Finally, pixel intensities are normalized in the range $[0; 255]$.

An experiment has been done to evaluate these three solutions: three different datasets have been obtained by converting the original event-based dataset applying the different polarity policies. Then the same ANN analyzed in Chapter 3, NxNet, has been trained on the three datasets, and depending on the accuracy achieved by the ANN on the test sets, it has been possible to select the best polarity policy.
The best solution has proved to be the third one, in which the polarity is discarded. The 2-channel accumulation solution has not shown particular improvements on the final accuracy, when compared to the case in which the polarity is discarded. At the same time, having two channels that separately store the polarity comes with a series of drawbacks: the size of the dataset and the dimension of the ANN increases. Moreover, the number of neurocores occupied by the converted SNN is higher, thereby also impacting on the latency of the system. For this reason, the 2-channel policy can be discarded. Considering the 1-channel polarity accumulation, the obtained results have shown a drop of accuracy ($\simeq -4\%$) with respect to the discarded polarity case: this can be addressed to the fact that this solution leads to having frames with a general high level of pixel intensities, being all initialized to a non-zero value, thereby leading to lower classification results.
For these reasons, in Table 4.1 only the results achieved without signed polarity accumulation are reported.

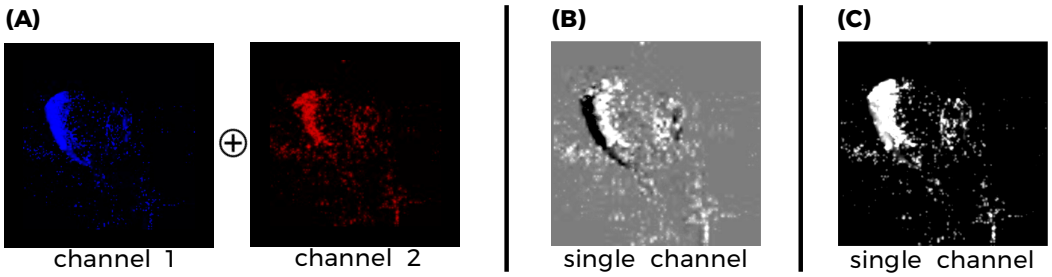


Figure 4.6: Polarity policies. **(A) 2 channel accumulation**: negative and positive events are accumulated on 2 different frame channels. **(B) 1 channel accumulation**: positive/negative events increment/decrement the pixel intensities of a single channel. **(C) Discard polarity**: events are accumulated on a single channel independently from their polarity.

4.3.4 Frame Size

Lastly, the dimension of the frame has to be chosen. The original recordings have a dimension of 128x128. However, such dimension may be too large when used as input of our converted SNN, leading to high number of neurocores needed to simulate the network, as well as increasing the latency of the prediction. Therefore,

we resized the image to a dimension of 32x32, applying two different techniques: **whole Window resize** and **Attention Window**.

Whole Window resize

Starting from the original 128x128 frame size, we reduce the frame dimension to 32x32 by applying a preliminary Average Pooling step. This process is also useful to remove noisy events present in the original recordings, producing an input frame that contains only the relevant gesture information. Also a 64x64 size reduction have been evaluated, but the accuracy results obtained by the ANN did not show any improvement over the 32x32 size. On the other hand, a size of 16x16 would be too small for achieving a good recognition by the ANN. A visual representation of this solution is reported in Figure 4.7(A).

Attention Window

Another solution, which has been proposed by [51] for the same dataset, is to collect only the events that are inside a 64x64 attention window, which moves and keeps track of the incoming gestures. Then, the Average Pooling is applied on the 64x64 frame, reducing its size to 32x32. This solution is shown in Figure 4.7(B).

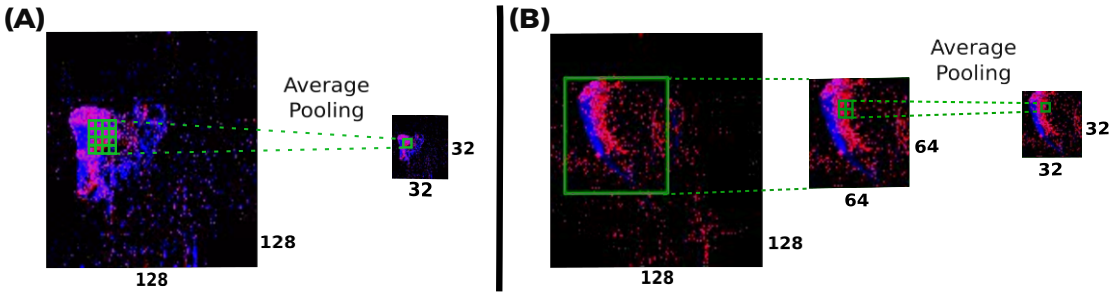


Figure 4.7: Frame resize policy. **(A)** The original frame with size 128x128 is scaled to a dimension of 32x32 using an Average Pooling filter. **(B)** An attention window of size 64x64 is extracted and resized to a dimension of 32x32 applying Average Pooling.

4.3.5 Dataset Structure

In all the above-discussed pre-processing approaches, the frames are associated to their corresponding labels, and accumulated into a **train set** and a **test set**. The dimension of the dataset depends on the chosen pre-processing approaches. Less frames are generated with longer time-windows, whereas the amount of frames increases as the time-window covered by each frame gets shorter.

4.4 ANN Accuracy Results

4.4.1 Experimental setup

All the obtained pre-processed datasets have been tested with the NxNet, the same ANN analyzed in Chapter 3, along with the same training parameters for the MNIST and CIFAR10 datasets. This choice has been made to ensure that the possible inconsistencies between the ANN and SNN accuracy results depends only on the data pre-processing stage, and are not related to the network architecture or the training policies. As explained in Section 3.6, if the ANN architecture is modified, the conversion process may suffer and the resulting SNN might not show the expected behavior.
The conversion process has been executed applying the soft reset mode, and a simulation duration of 256 time-steps, with a DThIR=2^1, since these are the settings that have shown the best results for both the MNIST and the CIFAR10 analyses.

4.4.2 Whole Window resize

Table 4.1 shows the accuracy results of the ANN on the different post-processed datasets. In all the reported cases, the size of the frame is set to 32x32, and the events polarity is discarded.

Experiment	duration(ms)	overlap	channels	ANN accuracy
E1	60 (10 per ch.)	✗	6	85.23%
E2	60 (20 per ch.)	✗	3	85.44%
E3	150 (50 per ch.)	✗	3	87.89%
E4	235 (78 per ch.)	✗	3	88.63%
E5	300 (100 per ch.)	✗	3	88.33%
E6	100	✗	1	74.14%
E7	235 (78 per ch.)	2	3	88.87%
E8	300 (100 per ch.)	2	3	**90.46%**

Table 4.1: Pre-processing techniques applied to the original gesture DVS dataset and relative ANN accuracies. In all the experiments, the frame size is equal to 32x32 and the polarity inormation is discarded. All the generated datasets have been tested with the NxNet ANN.

Experiment E1 shows that, choosing a time window of only 60ms gives low accuracy results, and the same goes for experiment E2, where the time range

covered by each channel is doubled: this can be tracked back to a few amount of events accumulated per channel.
Moving to experiments E3-5, the time window is progressively incremented, until a maximum duration of 300ms is covered. The results show that a good level of accuracy is reached with a 3-channel frame covering periods of 235ms and 300ms.
Experiment E6 has been realized to see if the use of a single channel frame could be a valid solution. In this case, the accuracy drop is evident, and this can be easily addressed to the fact that the single frame does not contain the temporal information, being all the events accumulated in a single channel.
Moving to experiments E7 and E8, an overlap factor equal to 2 is introduced. The accuracy in these two cases increases, reaching a value of **90.46%** in experiment E8, that is the best obtained value.

4.4.3 Attention window resize

As explained in section 4.3.4, another preprocessing approach is based on isolating the 64x64 region of the input frame where the gesture is happening, and then apply average pooling to get a 32x32 frame size. We have built a new dataset in which the *attention window* policy is followed. For what concerns the other preprocessing choices, the temporal window covered by each frame is 300ms, with each frame containing 3 channels each covering 100ms. The polarity information is discarded and the overlapping factor is set to 2.
This dataset, apart for the use of the attention window, is the same of experiment E8 from section 4.4.2, where the *whole window* resize policy is used. We have decided to compare the two policies (Attention Window vs Whole Window) in the best case scenario.
The test accuracy result achieved by NxNet on this *attention window*-based dataset is only **87.35%**, that is -3% with respect to the once achieved with a *whole window* resize policy in experiment E8 from Table 4.1.
In order to evaluate the reasons behind such poorer accuracy result, it is possible to analyze the confusion matrix of both the experiments, reported in Figure 4.8.

As one can see from the reported confusion matrices, in the *attention window* dataset case the ANN gets better results for the classes *air guitar* and *other gestures*. On the other hand, the *whole window* resize policy gets better recognition results for all the other 9 classes.
This results can be explained by assuming that the spatial relation between the gesture and the body position is an important information for the ANN to correctly recognize the gesture. In fact, when the attention window is applied, the gesture is taken out of context, leading to an inevitable information loss.

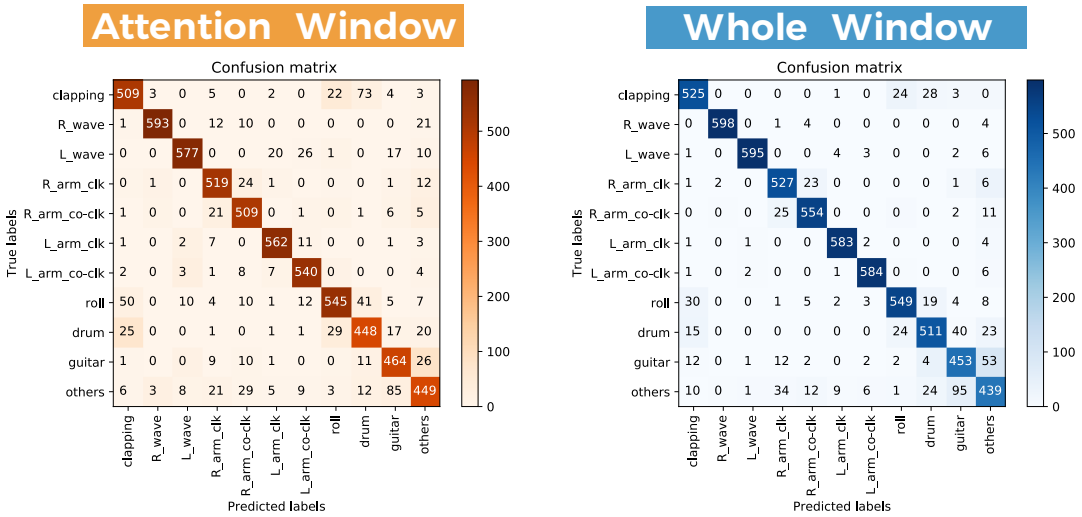


Figure 4.8: Comparison of the confusion matrices obtained from the attention window and whole window results on the NxNet

4.5 SNN inference

The trained NxNet model of experiment E8 is then converted to its equivalent SNN and tested on the Intel Loihi. The converted network reaches a test accuracy of **89.64%**, which is only 0.82% lower with respect to the original ANN. Moreover, the average time for classifying an input frame is **11.43ms**.
This results has to be compared with the state-of-the-art test accuracies achieved in [10] and in [37].
[10] reaches a test accuracy of 94.59% with a 64x64 frame size, whereas the accuracy achieved on a 32x32 frame drops down to 90.78%. This last value is only 1,14% higher than the one obtained in this research using frames with the same dimension of 32x32, but it is obtained with an ANN that is far bigger (16 convolutional layers with far more feature maps per layer) with respect to the one that we used. However, we did not consider to employ such large and deep networks, to maintain low resource utilization and low latency for a real-time system.
In [37] the test accuracy reached on a smaller portion of the original dataset (1.5 seconds per gesture) is 93.64%, that is 4% higher with respect to the one obtained with our methodology. However, being their network a pure SNN, they have directly used the original event-based dataset, avoiding an inevitable information loss that is related to the pre-processing step.
In terms of latency, with our best solution (E8) the total time needed for a frame classification is $150ms + 11.42ms = \mathbf{161.42ms}$. Since the overlap factor is 2, the next frame starts after 150ms, therefore we considered 150ms of accumulation time

per frame. Moreover, we are neglecting the pre-processing time needed to obtain the input frame. This configuration gives a throughput of 6.24 classified frames per second, which constitutes a feasible solution for a real-time application.

4.6 Conclusions

In this chapter, we have proposed an efficient method for deploying gesture recognition through a DVS camera on the Loihi neuromorphic processor. After a careful study of the original event-based DvsGesture dataset, we devised an efficient pre-processing method for accumulating the events into an equivalent frame-based dataset. As shown by our results, this process enables the training in the ANN domain. Therefore, the well-known training policies and optimizations for ANNs can be employed. An efficient conversion of the trained ANN into the SNN domain enables the accurate, energy-efficient and real-time processing on a neuromorphic embedded platform such as the Loihi.

Chapter 5

Spiking MobileNet implementation on Loihi

5.1 MobileNet Architecture

When it comes to the implementation of deep neural networks, the network architecture design has a major role in determining its computational cost and latency. An ANN gets deeper as we increase the number of layers, and wider as the number of feature maps per layer increases. This can lead to highly computationally intensive networks, that require a lot of hardware resources to be executed.
When dealing with limited hardware devices, like in embedded systems for mobile vision, we need to go for some optimization approaches that can reduce the network size and computational cost, still providing good accuracy results.
MobileNet [11] is an efficient network architecture that drastically reduces the number of parameters needed to build deep networks, providing an higher efficiency in terms of accuracy per computational cost.
The design of a MobileNet is based on the use of **Depthwise Separable Convolutions** (**DSCs**), that substitute traditional convolutional layers leading to a massive saving in terms of network parameters and number of computations required to execute convolution operations.

5.1.1 Depthwise Separable Convolution

In a classical convolutional layer, when the filter is applied on the input feature map two operations are executed simultaneously: first, each input channel is filtered applying the convolution operation, then the filter combines the information among the input channels providing an output feature map. On the other hand, **DSC** layers split this two operations into two subsequent steps [11]:

- First, the input image is filtered, reducing its size but keeping the same number of input channels. This operation is executed using **depthwise convolution** filters which operate individually on each input channel.
- As a second step, another convolutional operation called **pointwise convolution** is used to combine the pixels along the channel dimension.

This factorization allows to drastically reduce the number of operations that are needed to execute the entire convolution operation.
Taking in consideration a classical convolution filter, it takes as input a feature map with size $D_F \times D_F$ with M channels, and produces an output feature map with size $D_G \times D_G$ with N channels. Such filter has shape $D_K \times D_K \times M \times N$.
This lead to a computational cost in terms of number of needed multiplication equal to [11]:

$$D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F$$

On the other hand, in a DSC layer the filters used are two. First, **depthwise convolution** filters are applied on the input feature map: a total of M filters with size $D_K \times D_K$ are applied individually on each of the M channels of the input feature map. This produces an intermediate feature map with size $D_G \times D_G$ and M channels. Then, the **pointwise convolution** is executed by using N $1 \times 1 \times M$ filters, which combine the information along the input channels and provide an output feature map with size $D_G \times D_G$ with N channels. In this case, the number of multiplications needed is the sum of the multiplications of the first depthwise filter and the second pointwise filter [11]:

$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F$$

From this, we can compute the reduction of computation with respect to a single step convolution:

$$\frac{D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F}{D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F} = \frac{1}{N} + \frac{1}{D_K^2}$$

Therefore, the use of a DSC leads to a massive reduction in the number of multiplications, that is directly proportional to the number of output channels N and the filter square size D_K^2.
Figure 5.1 shows a visual comparison between a standard convolution and a depthwise separable convolution.
The use of DSC layers also reduces the number of parameters of the network [11]: Given a convolutional filter, its number of parameters is equal to $D_K \times D_K \times N \times M$. On the other hand, the number of parameters of a DSC is the sum of the depthwise convolution parameters and pointwise convolution parameters: $D_K \times D_K \times M + M \times N$.
This allows to build very deep networks keeping the size of the model very small when compared to equivalent networks implemented with traditional convolutions.

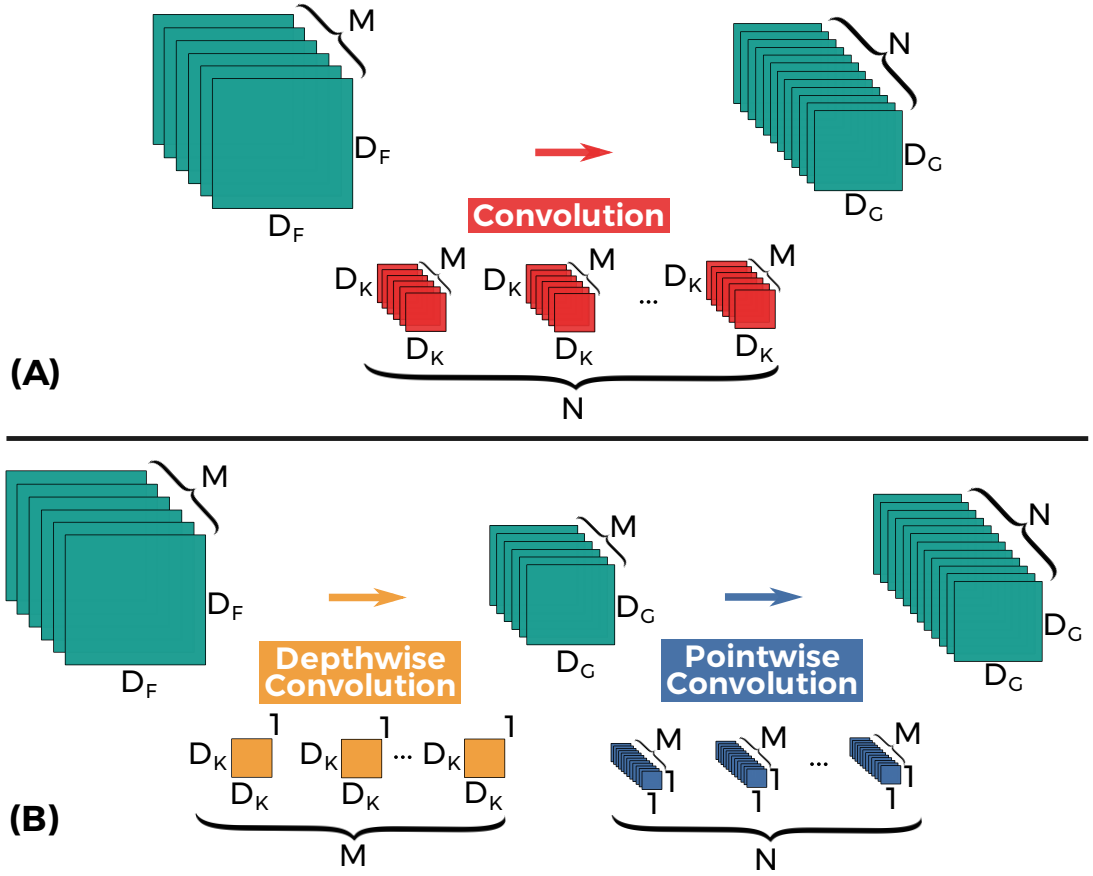


Figure 5.1: **(A)** Visual representation of a standard convolution. **(B)** Generic example of a depthwise separable convolution, containing both the depthwise convolution and pointwise convolution steps [Source: [11]].

5.1.2 Depthwise convolution on Loihi

When it comes to the implementation of Deep SNNs on Loihi, mapping large layers on the architecture neurocores may be non-trivial. This is mostly due to the limitations in the neurons fan-in and fan-out given by the neurocore constraints reported in Table 2.1. By using standard convolutions, the number of input synapses may be very large when the input feature map has many channels, being equal to $D_K \times D_K \times M$.
On the other hand, the use of DSCs can lead to a consistent reduction of neurons fan-in. In fact, by dividing the convolution in two subsequent steps, the number of input neurons reduces. Neurons that receives the output of the depthwise convolutions will have a number of input synapses equal to $D_K \times D_K$, and the following group of neurons placed after the pointwise convolution will receive a

total of $1 \times 1 \times M$ inputs per neuron. This drastically reduces the complexity of the converted SNN mapping on the Loihi neurocores, leading to a lower amount of used cores.
To better understand the fan-in reduction, let's look at the example reported in Figure 5.2. In this example we consider a 5x5 feature map with 3 channels on which it is applied a convolution with 3x3 kernel with stride 1. On the left side of the Figure, the convolution is performed in two steps, using a DSC layer. As one can see, the fan-in of neurons after the depthwise convolution (first step) is 9, whereas the fan-in after the pointwise convolution (second step) is only 3. On the other hand, as shown in the right side of the Figure, when a conventional convolution is applied, the fan-in of the output neurons grows significantly, reaching a value of 27 synaptic inputs. Not considering the high reduction in terms of multiplications and addictions that comes with the use of DSCs, the consistent reduction of the fan-in makes the use of these layers crucial for the development of deep neural networks on Loihi. To better understand the real benefits that comes from the use of DSCs,

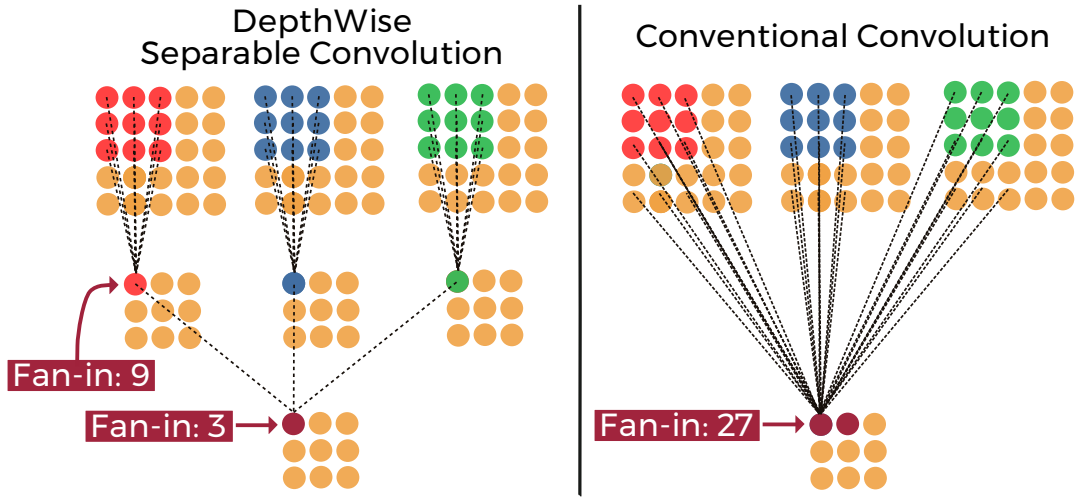


Figure 5.2: Fan-in of a depthwise separable convolution (left) and fan-in of a conventional convolution (right)

we can compare the number of neurocores occupied by two equivalent SNNs, one fully made of conventional convolutional layers, whereas the second one built with DSCs layers. The results of this comparison are reported in Table 5.1. The use of depthwise separable convolutions lead to a reduction of 485 neurocores, saving 78.5% of hardware resources.

Layer	filter	output shape	neurocores Conv2D	neurocores DSC
Input		32x32x3	6	6
Conv	4x4x32, stride 2	15x15x32	75	15
Conv	3x3x64, stride 1	15x15x64	225	45
Conv	3x3x128, stride 2	7x7x128	154	26
Conv	3x3x128, stride 1	7x7x128	112	28
Conv	3x3x256, stride 2	3x3x256	24	8
Conv	3x3x512, stride 1	1x1x512	18	4
flatten		512	/	/
Dense		10	1	1
		Total:	618	133

Table 5.1: Neurocores occupation comparison between a fully convolutional SNN and a corresponding depthwise-based convolutional SNN.

5.1.3 Network architecture

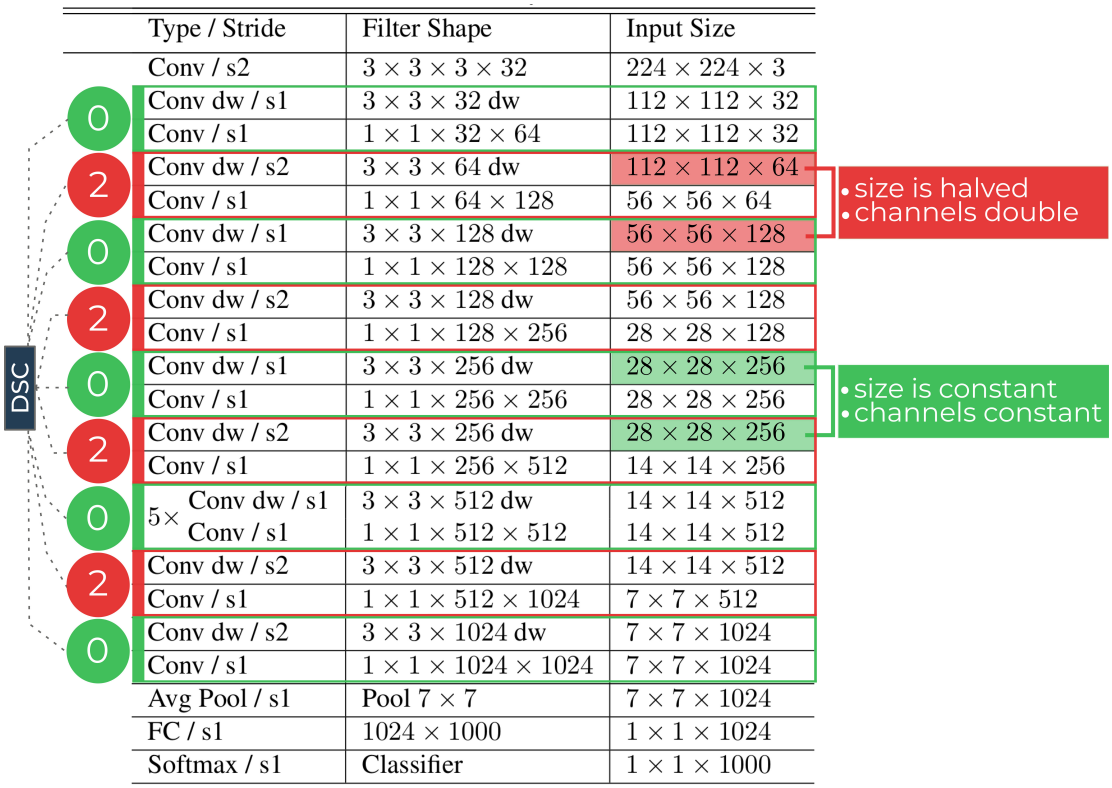
As a first step, let's start by analyzing the architecture of the original MobileNet. The network was originally designed for ImageNet [52], a very large dataset with 224x224 RGB images. The network structure is reported in Figure 5.3.

As one can see, besides a first conventional convolutional layer, the network is structured as a series of depthwise convolutions, that are divided into 2 types:

- **DSC0**: A depthwise separable convolution with 3x3 kernel and stride 1, which does not reduce the size of the input feature map and neither increases its number of channels. Both the inner depthwise and the pointwise convolutional layers are followed by a BatchNormalization layer, before applying the ReLU activation function on the outputs.

- **DSC2**: A depthwise separable convolution with 3x3 kernel and stride 2, which reduce by half the size of the input feature map and doubles the number of input channels. Both the inner depthwise and the pointwise convolutional layers are followed by a BatchNormalization layer before applying the ReLU activation function on the outputs.

It should be noted the use of 5 consecutive DSC0 blocks, that consistently increase the size of the whole network. Finally, the network comprise an AveragePooling layer, needed to reduce the size of the feature map, and a fully connected layer that generates the output predictions.
In order to see the full potential of this network in terms of accuracy, number of operations and parameters, we can compare it with an equivalent MobileNet that



	Type / Stride	Filter Shape	Input Size
	Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
0	Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
	Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
2	Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
	Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
0	Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
	Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
2	Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
	Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
0	Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
	Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
2	Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
	Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
0	$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
	Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
2	Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
	Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
0	Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
	Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
	Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
	FC / s1	1024×1000	$1 \times 1 \times 1024$
	Softmax / s1	Classifier	$1 \times 1 \times 1000$

Figure 5.3: Architecture of the original MobileNet [11].

uses conventional convolutional layers instead of DSCs. The comparison is reported in Table 5.2.

Model	Accuracy	Mult-Adds	Parameters
Conv MobileNet	71.7%	4866 (Million)	29.3 (Million)
MobileNet	70.6%	569 (Million)	4.2 (Million)

Table 5.2: Depthwise separable vs fully convolutional MobileNet

As one can notice, the use of depthwise separable layers comes with a minimal accuracy reduction of only 1.1%, yet drastically reducing the number of multiplications and additions needed, as well as the number of parameters by a factor 8.5x and 7x respectively.

5.2 Adapting the MobileNet to Loihi constraints

5.2.1 Pseudo-MobileNets

The main focus of this research is to evaluate the ANN-to-SNN conversion process of different pseudo-MobileNet networks, i.e., networks that adopt depthwise separable convolutions instead of conventional convolutions following the principles of the original MobileNet described in Section 5.1.3. The networks are built varying the use and order of DSC layers. As a consequence, it is possible to generate several networks that differ in terms of size and depth.
In order to understand the design policy adopted, it is necessary to draw some distinctions from the original MobileNet design. The original MobileNet has been designed to operated on ImageNet dataset, which contains very large images (224x224x3), as a consequence, the network is very big in size, and it is not suitable for a direct implementation on Loihi. In order to reduce the dimensions of the network, we choose to use another dataset, CIFAR10 [9], which contains far smaller (32x32x3) images. This allows to reduce the dimension of the network, making it possible to run the converted SNNs on Loihi.
Given the reduced dimensions of the dataset size, it is also necessary to modify the network structure. First of all, 3 different types of depthwise separable convolutions have been adopted. In addition to the layers **DSC0** and **DSC2**, which are present also in the original version of the MobileNet, we have added the layer **DSC1**: this depthwise separable layer reduces by 2 the size of the input feature map, doubling the number of channels. As for DSC0 and DSC2, in DSC1 both the depthwise and the pointwise convolutional layers are followed by a BatchNormalization layer, and then the ReLU activation function is applied on the outputs of both the intermediate layers.
The 3 DSC layers are summarized in Figure 5.4. The introduction of the layer DSC1 has been necessary in order to have a less significant size reduction of the feature maps with respect to DSC2, allowing to build deeper networks.

5.2.2 Design policy

Networks have been designed with several purposes:

- Build progressively deeper and larger networks, in order to have a clearer view on how the size of the original ANN translates in the number of Loihi neurocores occupied by the converted SNN.

- Find the converted SNNs with the best trade-off between accuracy and hardware occupation.

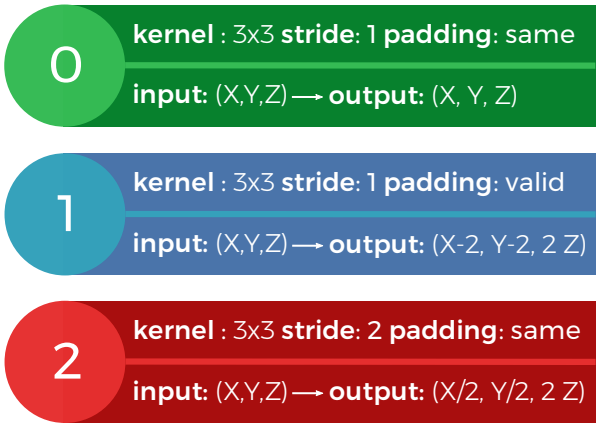


Figure 5.4: Depthwise separable convolutional layers adopted in our research.

- Analyze how the converted networks behave dependently on the layer used, as well as on the order and the number of layers adopted.

As a consequence of these principles, several networks have been realized. Networks differs in terms of depthwise separable layers adopted and number of convolutional filters applied. For all the networks, the first layer is in common: as like as in the original MobileNet, the first layer is a conventional convolutional layer: its kernel size is set to 4x4 with stride 2, with a total of 32 filters. The choice of using this kind of layer comes from the need to reduce the feature map size, in order to adopt the following depthwise separable layers on a smaller feature map with size 15x15. This solution is extremely important when it comes to implementing the SNNs on Loihi, because large feature maps may be difficult to be efficiently mapped on neurocores. The same approach is used in the original MobileNet to reduce the feature map size from 224x224 to 112x112.
Then, after the first convolutional layer, only depthwise separable convolutions layers are used. Each of the designed network is defined by the series of depthwise separable layers adopted, as well as by the feature maps used for each layer. As in the original MobileNet, the number of filters applied are chosen to be incremental, such that deeper layers have feature maps with an higher number of channels, as well as a smaller size. Depending on the type of depthwise separable filters used, the number of the output channels can be equal to the input channels, for **DSC0**, or double than the input channels, as for **DSC1** and **DSC2**. As we will see in Section 5.2.3, this channel progression policy has been adopted for the majority of the implemented networks. However, in some cases it has been necessary to avoid the channel doubling for layers DSC1 in order to avoid to have extremely large feature maps in the last layer of the network.

5.2.3 Experimental setup

Implemented networks

Network models can be identified by the order of depthwise filters applied: for example, model 02021 will be structured as:

- First, there is always a **conventional convolution** with $kernelsize = 4x4$ and $stride = 2$.
- a sequence of 5 depthwise separable convolutions: **DSC0**, **DSC2**, **DSC0**, then another **DSC2** and finally **DSC1**.
- The last layer is always a fully connected **dense** layer with 10 output neurons with a *softmax* activation function applied.

A detailed visual representation of this network, is reported in Figure 5.5.
All the networks are designed to have as output of the last depthwise separable convolution a feature map with size (1,1) or, at most, (2,2). This design choice comes from the need of having a small feature map before the final fully connected layer. Otherwise, the number of synapses needed to connect the last convolutional layer and the following dense layer would be very high, leading to a very high fan-in, and consequently to the possibility of not being able to fit the last layer into the Loihi neurocores.
A complete report of all the networks that have been designed for this research is presented in Table 5.3. For each model, only the depthwise separable layers progression is reported. Each model has been designed such that it has as first layer a conventional convolution with 32 filters, as reported in 5.2.1. Moreover, each network ends with a fully connected dense layers that comprise 10 output neurons. For each model, represented by the order of its layers, two possible feature map channel progression are adopted: in the first case (**A**), the first depthwise layer start with 32 filters, whereas in the second case (**B**) the first depthwise separable convolution comprises 64 filters. Then, the filter progression depends on the DSCs applied, following the policy reported in 5.2.1. However, in the case of larger networks, it has been decided to avoid the doubling of the number of filters for layer DSC1 in the last positions. This has been done in order to avoid very large feature maps in the last layer, that would eventually lead to a difficult mapping of the network on Loihi.

ANN training and ANN-to-SNN conversion

The set of obtained pseudo-MobileNets ANNs are trained following the same principles adopted in Section 3.4.1, and then converted into equivalent SNNs. The number of training epochs is set to 135, that has shown to be a valid solution for

DSC layers	feature maps channel progression
221A	32 64 128
221B	64 128 256
1211A	32 64 128 256
1211B	64 128 256 512
2111A	32 64 128 256
2111B	64 128 256 512
02021A	32 64 64 128 256
02021B	64 128 128 256 512
11211A	32 64 128 256 512
11211B	64 128 256 512 1024
010211A	32 64 64 128 128 256
010211B	64 128 128 256 256 512
020111A	32 64 64 128 128 256
020111B	64 128 128 256 256 512
0101211A	32 64 64 128 128 128 256
0101211B	64 128 128 256 256 256 512
010101121A	32 64 64 128 128 256 512 512 1024
010101121B	64 128 128 256 256 512 1024 1024 2048

Table 5.3: Set of all the pseudo-MobileNet models proposed in the research.

all the proposed model in order to avoid overfitting. The conversion process is implemented following the results obtained in Section 3.5, setting the conversion parameters in order to achieve the best conversion results. More specifically, the **reset mode** adopted is *Soft Reset*, the **DThIR** is set to 2^1 and finally the **simulation duration** is set to 256 algorithmic time-steps.

5.3 Results

5.3.1 Model 02021B analysis

As it has been shown in Chapter 3, Section 3.5, there is always an accuracy gap between the original ANN model and the corresponding converted SNN. This gap can vary, depending on the network model, and on other factors:

- Trained weights and biases distributions: as explained in Section 3.2, the distribution of weights and biases may play a key role in the final conversion result. This is due to the possible presence of strong outliers in the distributions that may impact negatively on the quantization process that is performed

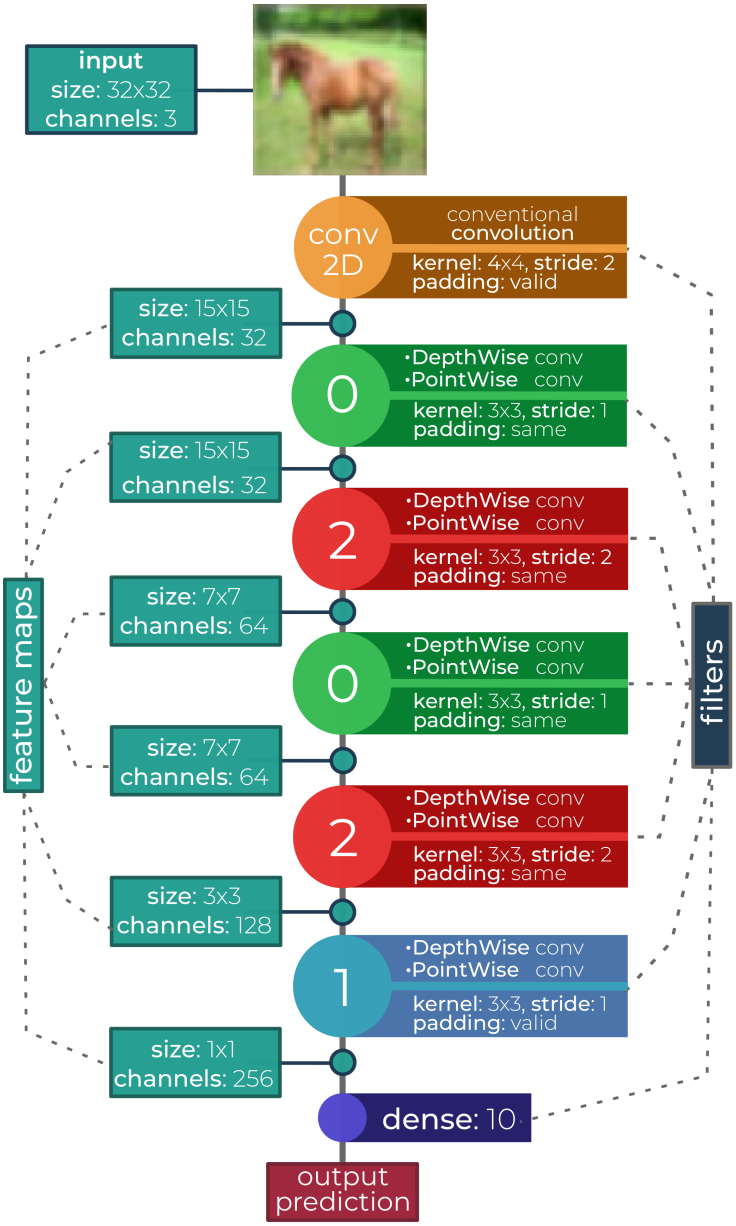


Figure 5.5: Example of a pseudo-MobileNet. The sequence of depthwise separable layers adopted is 02021. The first convolutional layer has 32 filters, and the following layers use a number of filters that follows the described pattern.

during the network conversion.

- SNN design: some specific issues may be related to the networks model itself. Very large networks may fail to correctly operate in the SNN domain, because

of a very high number of neurons per layer.

In order to evaluate the reasons behind failed conversions, let's focus on a specific case study, model $02021B$.
The analysis conducted follows the principles reported in 5.2.3. First, as always, we execute the training of the ANN model, applying a weight regulazation of 10^{-4} to each layer. As we will see in Section 5.3.2, this value is a good choice for this model setup. Once the training is completed, the model can be converted using SNN-ToolBox, and finally the converted SNN can be tested directly on Loihi.
After the training, the ANN accuracy achieved by the model is equal to 84.10%. However, the converted SNN gets an accuracy of only 53.92% on the test set. This means that the conversion process is not efficient, leading to an accuracy drop of 30.18%.
In order to understand the causes of such an accuracy drop, we can analyze the **correlation plots** of of each network layer. Correlation plots have been introduced in Section 3.2 as a valid tool to evaluate the conversion process. In figure 5.6 are shown the correlation plots of all the depthwise separable layers in the network. Moreover, for each layer is reported the value of the *Pearson Coefficient* (PC), that is an index that measures the linear correlation between two variables, i.e. activation and spikerates. The $PC \in [0,1]$, where 1 represent a perfect correlation. Each layer comprises one depthwise convolution and a subsequent pointwise convolution. As explained in Section 3.2, a correlation plot shows the correlation between the ANN neuron activation and the corresponding SNN neuron spikerates. In an ideal conversion we expect to have all the points aligned along the diagonal with a $PC = 1$, indicating that the conversion has been performed without information loss between the ANN and the SNN model. However, The reported results show that, as we move toward deeper layers of the network, the point distribution in the correlation plot gets wider, indicating that the converted spiking neurons are not behaving as expected. When the last depthwise separable layer (**DSC1**) is reached, the point distributions in the correlation plots get extremely sparse with $PC = 0.679$ and $PC = 0.432$ in the last depthwise and pointwise layers, leading to an evident inequality between the two models.

As previously discussed, the conversion inefficiency may come from a bad weights and biases quantization that takes place during the conversion process. Therefore, we can plot the weight and bias distributions of all the depthwise layers to see if there are discrepancies between the original ANN weight and bias distributions and the corresponding quantized distributions of the final SNN model. Figure 5.7 reports, for each depthwise separable layer, the weight and bias distributions of both the original ANN layer and the equivalent SNN. Moreover, for all the DSCs, both the depthwise and the pointwise convolutions distributions are shown.
Results show that the weights and biases distributions shapes are preserved for all the layers. However, in some cases the quantized biases distribution get squeezed

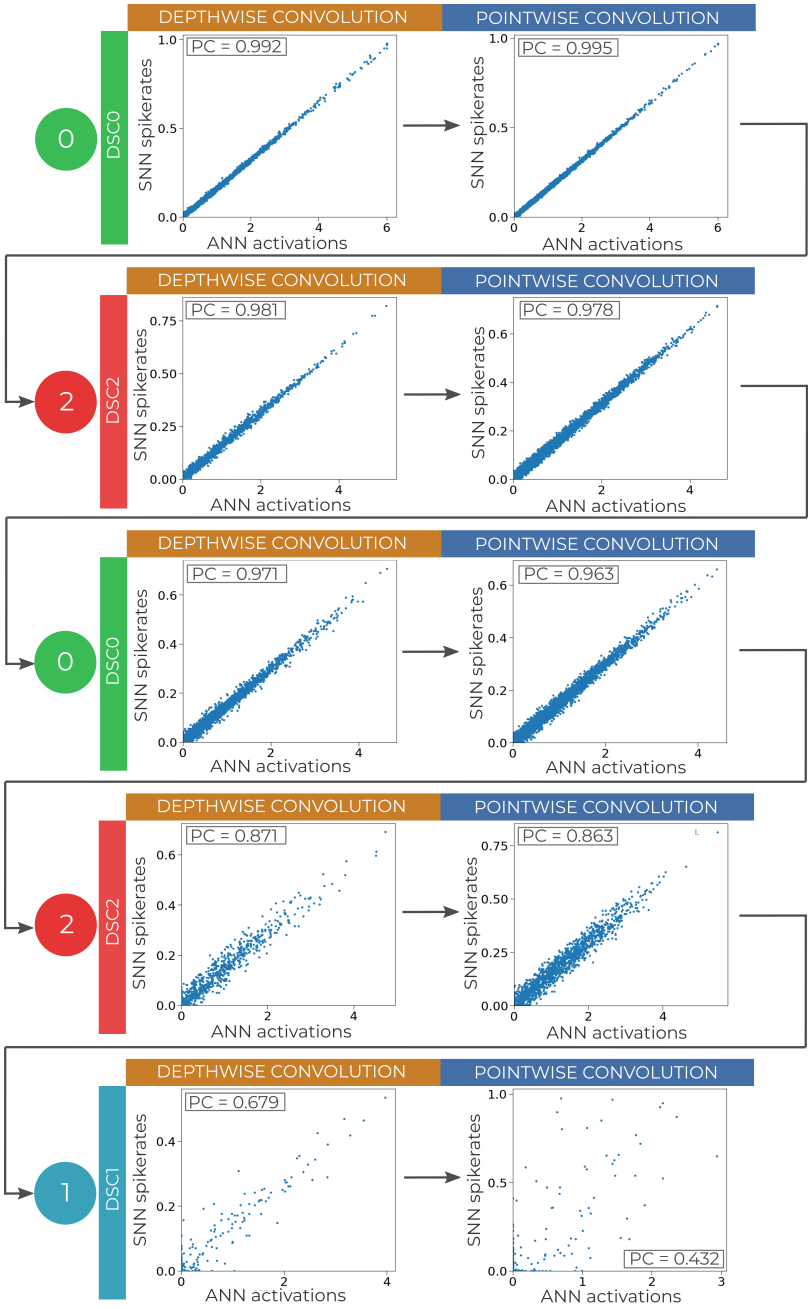


Figure 5.6: Correlation plots of model 02021B. For reason of clarity, only the depthwise separable layers are reported, while are omitted the correlation plots of the first convolution and the final dense layer. For each DSC, there are 2 inner layers: the depthwise convolution and the pointwise convolution.

into a smaller interval with respect to the original ANN distribution. Such a bias magnitude reduction is a consequence of the normalization process that takes place during the conversion. As explained in 3.1.3, during the conversion a normalization operation is executed in order to avoid the saturation of the neuron spikerates. This process requires to apply a scaling factor for weights, and also the membrane threshold is scaled to have a fixed *threshold to input ratio*. As a consequence of the threshold scaling, also biases need to be scaled by an additional factor proportional to the reduction applied to the membrane threshold of the previous layer.

In order to evaluate the possible influence of biases distributions on the converted network, we train again the ANN model 02021B removing the bias contributions from each convolutional layer. Then, the converted SNN is evaluated again by analyzing its correlation plots, that are reported in Figure 5.8.

As it can be clearly noticed, the correlation plots are far more accurate in this case. For almost all convolutional layers the correlation points are tightly distributed along the diagonal, with Pearson coefficients that are steadily higher than 0.95 for all layers except for the last pointwise convolution. These results clearly show that the bias contribution is responsible for the final results of the conversion.
However, besides the correctness of the correlation plots, we still need to evaluate both the ANN test accuracy and the final SNN test accuracy in order to see if the converted model works as expected. The results are reported in Table 5.4, that contains a comparison between the model 02021B trained both with and without biases.
In the case of no bias used, the model ANN test accuracy is equal to 78.87%, that is more than 5% lower than the one achieved by the original 02021B ANN using biases (84.10%). Moreover, the accuracy achieved by the converted SNN gets to a mere 40.40%, that is lower than the one obtained with the original 02021B model. Looking also at the accuracy difference between the ANN and the SNN models, when biases are removed from the convolutional filters the gap goes up to a 38.47%, that is far 8% higher with respect to the one of the original model with biases.

Model 02021B	**ANN accuracy**	**SNN accuracy**	**Accuracy Gap**
With Bias	84.10%	53.92%	30.18%
Without Bias	78.87%	40.40%	38.47%

Table 5.4: Recap of the accuracy results achieved with model 02021B. The first row reports the results achieved when using biases during training, whereas the second row shows the accuracy obtained with the same model trained without biases for all the convolutional layers.

These results show that the SNN conversion efficiency is not only related to the

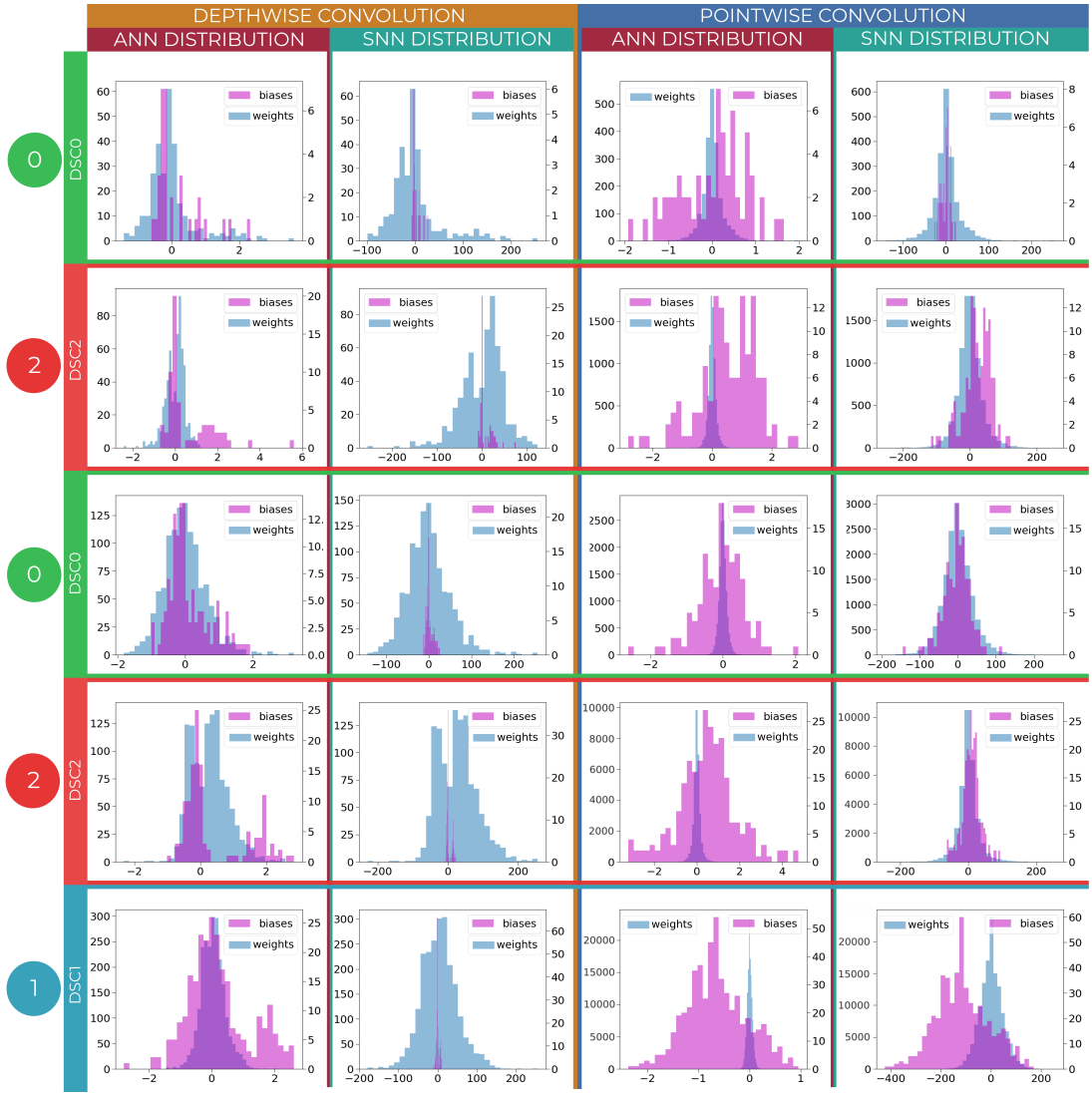


Figure 5.7: Weight and bias distributions of all depthwise separable layers for the model 02021B. For each layer, both the depthwise convolutional and the pointwise convolutional weights and biases distributions are displayed. Moreover, for each layer are reported both the original ANN distributions and the post-quatizations distributions of the converted SNN.

weights and biases conversion. More probably, the cause for a faulty conversion has to be found in the layer succession and the network architecture itself.

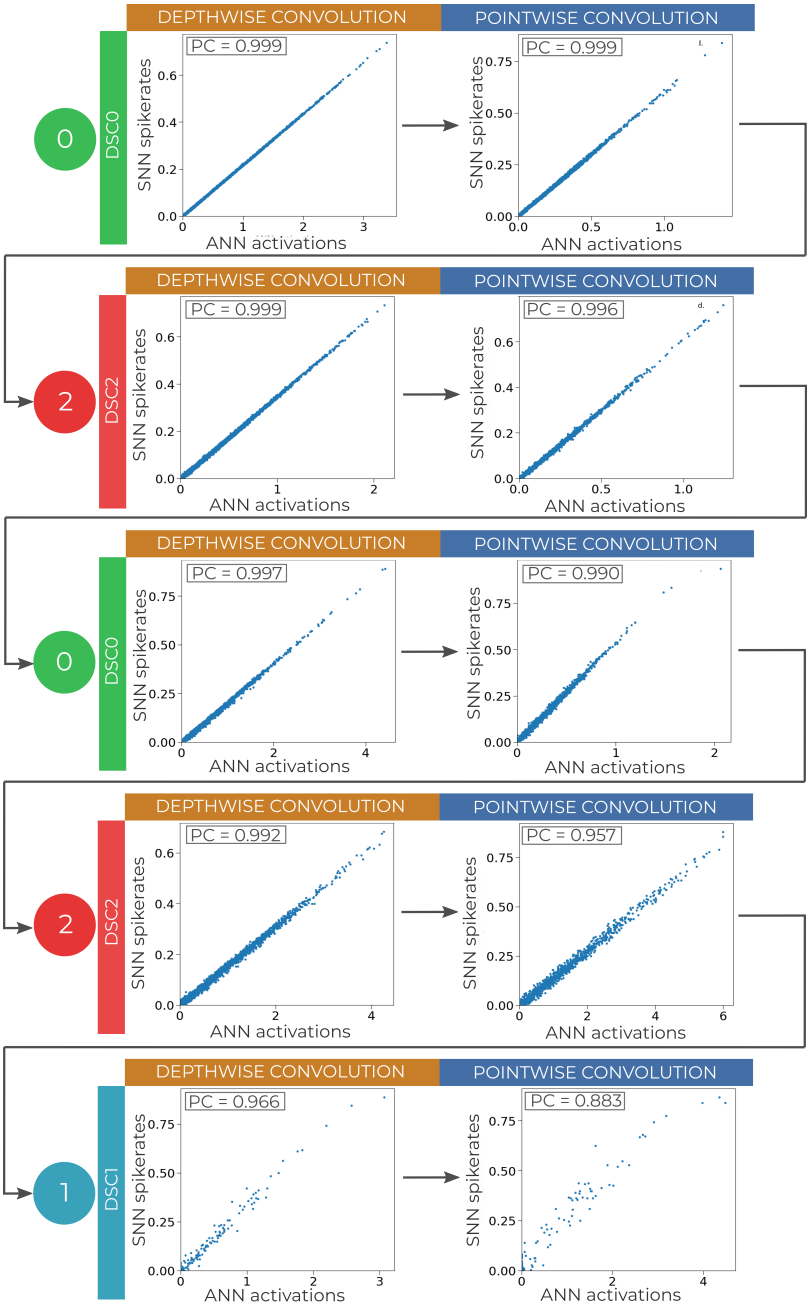


Figure 5.8: Correlation plots of model 02021B trained without bias. For reason of clarity, only the depthwise separable layers are reported, while are omitted the correlation plots of the first convolution and the final dense layer. For each DSC, there are 2 inner layers: the depthwise convolution and the pointwise convolution.

5.3.2 Weight Regularization analysis

As explained in 3.4.1, the use of **Weight Regularization** (WR) during training can help to achieve a better conversion output. In fact, this technique allows to reduce the possibility of having very large weights at the end of the ANN training, keeping the weight distribution away from the presence of strong outliers.
Therefore, **L2Regularization** [24] has been applied to all the ANNs layers, with 3 possible values: 10^{-4}, 10^{-3} and finally 10^{-2}. As this parameter increases, the regularization applied gets more strict.

The analysis conducted follows the principles reported in 5.2.3. First, all the ANN models reported in Table 5.3 are trained. Once the training is completed, the model can be converted using SNN-ToolBox, and finally the converted SNN can be tested directly on Loihi.

First of all, we can analyze the Loihi neurocores occupation for each of the converted SNNs. Figure 5.9 shows the incremental use of neurocores by the different SNNs. As expected, the number of neurocores occupied by the SNN increases with the number of layers, as well as with the number of channels of the feature maps. Models with filters progression **A** occupy always less than the corresponding **B** model. The results for model $010101121B$ is not reported because the network is too deep, and the mapping process fails. This sets an upper limit in our design space in terms of networks size.

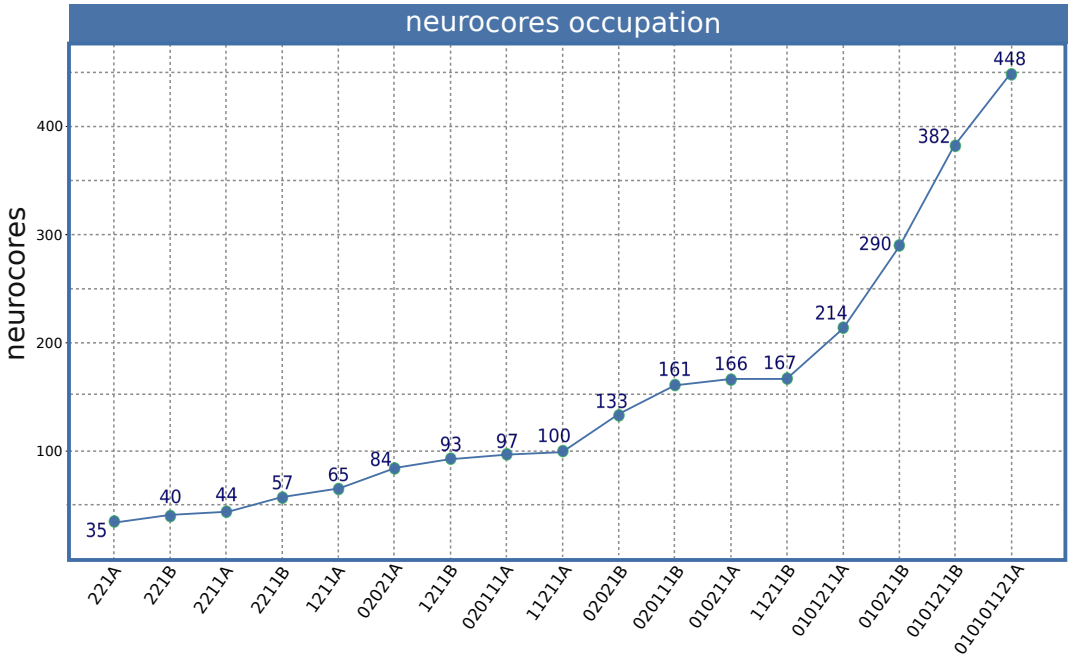


Figure 5.9: Neurocores occupation for each of the converted SNNs.

The accuracy results both for the ANN models and the corresponding converted SNNs are reported in Figure 5.10. For each weight regularization policy, are reported the ANN accuracy and the SNN accuracy of all the models under test. Moreover, the accuracy difference between the ANNs and the corresponding SNNs are highlighted. Models are organized in ascending order of neurocore occupation, following the results shown in Figure 5.9.
A general consideration is that a not negligible accuracy difference between the original ANN model and the converted SNN is present for several networks. Sometimes, this difference is low ($< 3\%$) but in many cases the gap is consistent, making the SNN very inefficient.
Overall, the worst results are given for the case of $WR = 10^{-2}$: for almost all the proposed networks, the ANN accuracy is lower with respect to the one achieved with the 2 other WR settings. As a consequence, also the SNN accuracies are lower with respect to the cases with $WR = 10^{-3}$ and $WR = 10^{-4}$.
Moving to the cases $WR = 10^{-3}$ and $WR = 10^{-4}$, in terms of SNN accuracy the results achieved differs with the models sizes. For small models, i.e. up to model $02021A$ excluding model $2211B$, an higher accuracy is reached by applying a more severe regularization (10^{-3}). On the other hand, for all the following deeper models an higher accuracy is achieved when a weaker regularization (10^{-4}) is chosen.
Looking at the accuracy differences between the ANN and SNN models, it is possible to notice that the case of $WR = 10^{-4}$ gives the better results for almost all the models. It is important to notice that this accuracy difference can be seen as a meter of the conversion quality: therefore, choosing a less strict regularization seems to be a better solution for the development of large spiking MobileNets.

5.3.3 Pareto Optimal Solutions

Given the results reported in Figure 5.9 and 5.10, it is possible to extract optimal models. We are trying to optimize two parameters at the same time: the hardware usage, expressed in terms of number of neurocores needed to map the network, and the test accuracy of the final SNN. This multi-objective optimization is usually referred to as **Pareto optimization** [53]. This solutions exploration does not provide a unique optimal model, but a set of Pareto optimal configurations that score the best with respect to both the accuracy and hardware usage. These optimal solutions belong to the Pareto-frontier curve.

Figure 5.11 shows the SNN accuracies and neurocores usage of all the models. More specifically, for each model the reported SNN accuracy is the best one among the three values obtained from the different weight regularization simulations. The faded results belong to models that are not optimal, having an accuracy lower than the one obtained by a competitive model that occupies a lower amount of

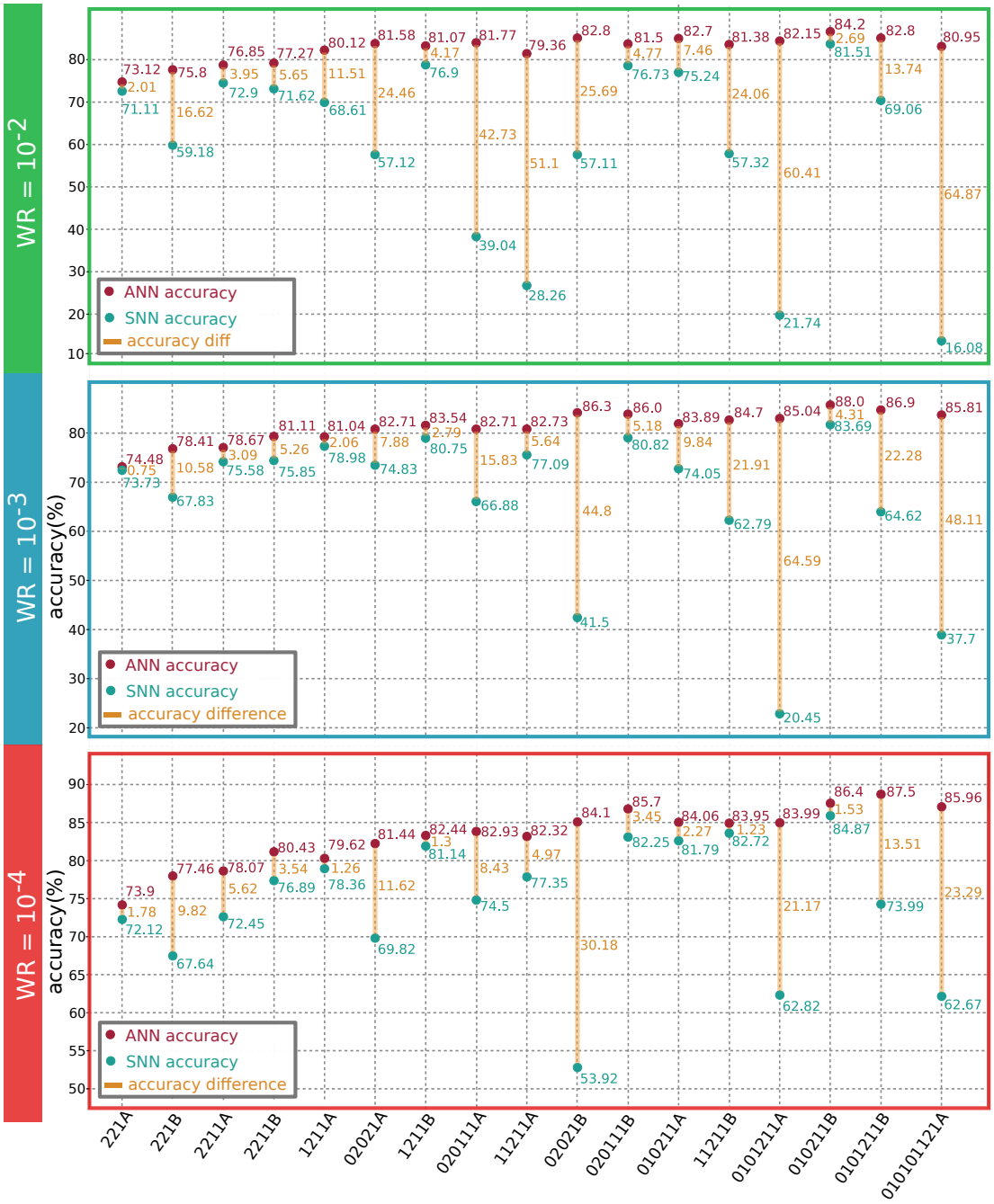


Figure 5.10: ANNs and SNNs test accuracies of all the implemented models. ANN and SNN accuracy results varies with the applied weight regularization, as well as with the model depth.

neurocores.
On the other hand, the orange curve highlights the networks that minimize the core occupation while maximizing the accuracy.

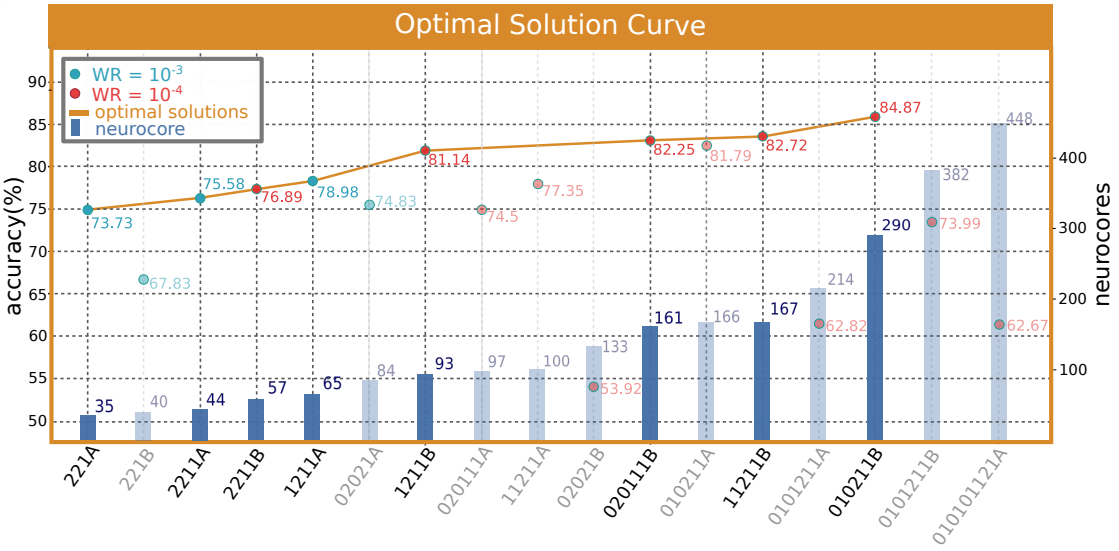


Figure 5.11: ANNs and SNNs test accuracies of all the implemented models and relative neurocores occupation. ANN and SNN accuracy results varies with the applied weight regularization, as well as by the model dept. The orange line is the Pareto-front set of solutions.

As results has shown, some network structures are more efficiently convertible, whereas some others are not equally efficient once transformed into equivalent SNNs.

5.4 Conclusions

The ANN-to-SNN conversion of pseudo-MobileNet models can lead to succesful results in both the final SNN accuracy as well as in the final Loihi usage.
For some network configurations there is still a gap between the original ANN and the final SNN accuracy results: in some cases, the SNNs can still provide good accuracy performances, whereas for other networks the gap is so high that the SNN can not be compared with the original ANN.
An in-depth analysis of one of the faulty conversions has shown that there is not a trivial solution to this problem. As it has been shown, the inefficiency of the final SNN can not be related exclusively to the conversion of weights and biases, and it is more likely related to the network structure itself.
An extensive analysis has been carried out in order to evaluate the conversion

of different models and find the network configurations that can maximize the final SNN accuracy while reducing the hardware usage. In this way, a set of Pareto-optimal solutions have been identified. As expected, deeper models can achieve higher accuracies, still occupying an higher amount of neurocores.

Chapter 6

Conclusions

Spiking Neural Networks represent a breakthrough in Artificial Intelligence, setting a new paradigm for building powerful yet efficient cognitive systems. The strongly biologically inspired nature of these models can lead to a more efficient information processing, as well as reducing the power consumption needed to execute complex tasks.
At the same time, the development of neuromorphic hardware is crucial for a step forward of SNNs. The brain-inspired design allows to efficiently simulate neurons properties and build large scale networks simulations. This, combined with the spiking-based computation, can lead to fast, extremely power efficient solutions.
In this research, the Intel Loihi neuromorphic processor has been extensively analyzed with the aim of building deep SNNs. Regarding the methodology used to train and build the networks, we have chosen to follow the ANN-to-SNN approach. This solution is the most promising for the development of very deep networks, combining the state-of-the-art solutions for ANN training, with the power efficiency and real-time behavior of the spiking domain.
However, the conversion process comes with challenges that are directly related to the inevitable differences among the two networks domains. Moreover, the conversion process has to face additional limitations when the SNN has to run on a limited precision hardware as Loihi.
As a consequence, we have produced an in-depth analysis of the full ANN-to-SNN conversion procedure on Loihi. Starting from the ANN training optimization, we have then evaluated the different parameters that play a role in the conversion process, in order to determine the setup that can provide efficient SNNs.
Then, we have moved to an application domain, realizing an efficient conversion process to convert spiking event data coming from a DVS camera and generate frame-based equivalent input. More specifically, we have focused on the optimization of the conversion process for the event-based IBM DvsGesture dataset, in order to produce an equivalent, ANN-compatible, frame-based dataset. Then, the converted

dataset has been used to train an ANN, that has then been converted into an SNN in order to provide an efficient SNN for gesture recognition.
Finally, we have focused on the development of deep SNNs, specifically dealing with MobileNet-inspired networks. Such model has proven to be an extremely good choice when it comes to implementing large networks on Loihi, matching the neurocores constraints of the device, such as fan-in and fan-out limitations and maximum number of neurons per core. However, in some network configurations there is still an evident accuracy gap between the original ANN model and the equivalent converted SNN. Such a difference is the consequence of an inefficient conversion. However, there is not a clear reason behind this inefficiency, and it may be directly related to the network architecture itself.
As a last mention, it is worth remarking how profound and revolutionary this new technology field can be. Building brain-inspired devices, working both on the hardware and software level, is the right path toward the development of extremely low-power, real-time cognitive systems.
We can learn so much on how to realize efficient, parallel devices by looking at the best computer ever made, *our brain*.

Bibliography

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. «Deep Residual Learning for Image Recognition». In: *CoRR* (2015) (cit. on p. ii).

[2] Davide Zambrano and Sander M. Bohte. «Fast and Efficient Asynchronous Neural Computation with Adapting Spiking Neural Networks». In: *CoRR* (2016) (cit. on p. ii).

[3] Paul A. Merolla et al. «A million spiking-neuron integrated circuit with a scalable communication network and interface». In: *Science* (2014) (cit. on pp. ii, 2, 22).

[4] Steve B. Furber, Francesco Galluppi, Steve Temple, and Luis A. Plana. «The SpiNNaker project». In: (2014) (cit. on pp. ii, 2, 22).

[5] Mike Davies et al. «Loihi: A Neuromorphic Manycore Processor with On-Chip Learning». In: (2018) (cit. on pp. ii, 2, 22–27).

[6] P. Lichtsteiner, C. Posch, and T. Delbruck. «A 128 X 128 120db 30mw asynchronous vision sensor that responds to relative intensity change». In: *ISSCC*. 2006 (cit. on pp. ii, 2, 28, 29).

[7] Bodo Rueckauer, Iulia-Alexandra Lungu, Yuhuang Hu, Michael Pfeiffer, and Shih-Chii Liu. «Conversion of Continuous-Valued Deep Networks to Efficient Event-Driven Networks for Image Classification». In: *Frontiers in Neuroscience* () (cit. on pp. ii, 21, 22, 27, 31–33, 36, 38).

[8] Yann LeCun, Corinna Cortes, and CJ Burges. «MNIST handwritten digit database». In: *ATT Labs [Online]* 2 (2010) (cit. on pp. iii, 10, 37).

[9] Alex Krizhevsky. *Learning multiple layers of features from tiny images*. Tech. rep. 2009 (cit. on pp. iii, 10, 37, 63).

[10] A. Amir et al. «A Low Power, Fully Event-Based Gesture Recognition System». In: *CVPR*. 2017 (cit. on pp. iii, 45, 46, 55).

[11] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. «MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications». In: *CoRR* (2017) (cit. on pp. iii, 4, 57–59, 62).

[12] *Google Unveils Neural Network with "Superhuman" Ability to Determine the Location of Almost Any Image*. 2016. URL: https://www.technologyreview.com/s/600889/google-unveils-neural-network-with-superhuman-ability-to-determine-the-location-of-almost/ (cit. on p. 1).

[13] *what are the machine learning use case in it operations?* 2019. URL: https://blog.imarticus.org/what-are-the-machine-learning-use-case-in-it-operations-machine-learning-blog/ (cit. on p. 1).

[14] J. B. Heaton, Nicholas G. Polson, and J. H. Witte. «Deep Learning in Finance». In: *CoRR* (2016) (cit. on p. 1).

[15] *Ascent of machine learning in medicine*. 2019. URL: https://www.nature.com/articles/s41563-019-0360-1 (cit. on p. 1).

[16] *10 Ways Machine Learning Is Revolutionizing Marketing*. 2018. URL: https://www.forbes.com/sites/louiscolumbus/2018/02/25/10-ways-machine-learning-is-revolutionizing-marketing/#34516bb05bb6 (cit. on p. 1).

[17] Chetan Singh Thakur et al. «Large-Scale Neuromorphic Spiking Array Processors: A quest to mimic the brain». In: *CoRR* (2018) (cit. on p. 1).

[18] N. Jouppi, C. Young, N. Patil, and D. Patterson. «Motivation for and Evaluation of the First Tensor Processing Unit». In: *IEEE Micro* (2018) (cit. on pp. 1, 22).

[19] Nikola K. Kasabov. In: *Time-Space, Spiking Neural Networks and Brain-Inspired Artificial Intelligence*. Springer-Verlag Berlin Heidelberg, 2019 (cit. on pp. 1, 16).

[20] Sebastian Schmitt et al. «Neuromorphic hardware in the loop: Training a deep spiking network on the BrainScaleS wafer-scale system». In: *2017 International Joint Conference on Neural Networks (IJCNN)* (2017) (cit. on p. 2).

[21] *Building a Silicon Brain*. 2019. URL: https://www.the-scientist.com/features/building-a-silicon-brain-65738 (cit. on p. 3).

[22] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016 (cit. on pp. 6–8, 12, 14).

[23] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. «Dropout: A Simple Way to Prevent Neural Networks from Overfitting». In: *Journal of Machine Learning Research* (2014) (cit. on pp. 12, 13).

[24] Corinna Cortes, Mehryar Mohri, and Afshin Rostamizadeh. «L2 Regularization for Learning Kernels». In: *CoRR* (2012) (cit. on pp. 12, 34, 73).

[25] Diederik Kingma and Jimmy Ba. «Adam: A Method for Stochastic Optimization». In: *International Conference on Learning Representations* (2014) (cit. on pp. 13, 37).

[26] Sebastian Ruder. «An overview of gradient descent optimization algorithms». In: *arXiv preprint arXiv:1609.04747* (2016) (cit. on p. 13).

[27] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. «On the importance of initialization and momentum in deep learning». In: Proceedings of Machine Learning Research. 2013 (cit. on p. 13).

[28] Shorten Connor and Khoshgoftaar Taghi M. «A survey on Image Data Augmentation for Deep Learning». In: *Journal of Big Data* (2019) (cit. on p. 13).

[29] *A Comprehensive Guide to Convolutional Neural Networks*. 2019. URL: `https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53` (cit. on pp. 14, 15).

[30] *Deep Learning and Convolutional Neural Networks*. URL: `https://www.rsipvision.com/exploring-deep-learning/` (cit. on p. 16).

[31] Michael Pfeiffer and Thomas Pfeil. «Deep Learning With Spiking Neurons: Opportunities and Challenges». In: *Frontiers in Neuroscience* (2018) (cit. on pp. 16, 19).

[32] Nguyen Phong, Masoud Daneshtalab, Sergei Dytckov, Juha Plosila, and Hannu Tenhunen. «Silicon synapse designs for VLSI neuromorphic platform». In: *NORCHIP 2014 - 32nd NORCHIP Conference: The Nordic Microelectronics Event* (2015) (cit. on p. 17).

[33] Wulfram Gerstner and Werner M. Kistler. In: *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002 (cit. on pp. 17, 18).

[34] Filip Ponulak and Andrzej Kasiński. «Introduction to spiking neural networks: Information processing, learning and applications». In: *Acta neurobiologiae experimentalis* (2011) (cit. on p. 20).

[35] Gopalakrishnan Srinivasan, Priyadarshini Panda, and Kaushik Roy. «STDP-Based Unsupervised Feature Learning Using Convolution-over-Time in Spiking Neural Networks for Energy-Efficient Neuromorphic Computing». In: *J. Emerg. Technol. Comput. Syst.* (2018) (cit. on pp. 20–22).

[36] Bodo Rückauer, Nicolas Känzig, Shih-Chii Liu, Tobi Delbrück, and Yulia Sandamirskaya. «Closing the Accuracy Gap in an Event-Based Visual Recognition Task». In: *CoRR* (2019) (cit. on pp. 21, 22, 47).

[37] Sumit Bam Shrestha and Garrick Orchard. «SLAYER: Spike Layer Error Reassignment in Time». In: *Advances in Neural Information Processing Systems 31*. 2018 (cit. on pp. 21, 55).

[38] E. O. Neftci, H. Mostafa, and F. Zenke. «Surrogate Gradient Learning in Spiking Neural Networks: Bringing the Power of Gradient-Based Optimization to Spiking Neural Networks». In: *Signal Processing Magazine* (2019) (cit. on p. 21).

[39] Maxence Bouvier et al. «Spiking Neural Networks Hardware Implementations and Challenges: A Survey». In: *ACM Journal on Emerging Technologies in Computing Systems* (2019) (cit. on p. 22).

[40] Trevor Bekolay, James Bergstra, Eric Hunsberger, Travis DeWolf, Terrence Stewart, Daniel Rasmussen, Xuan Choo, Aaron Voelker, and Chris Eliasmith. «Nengo: a Python tool for building large-scale functional brain models». In: *Frontiers in Neuroinformatics* (2014) (cit. on p. 27).

[41] Marcel Stimberg, Romain Brette, and Dan Goodman. «Brian 2, an intuitive and efficient neural simulator». In: *eLife* (2019) (cit. on p. 27).

[42] Andrew Davison, Daniel Brüderle, Jochen Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger. «PyNN: A Common Interface for Neuronal Network Simulators». In: *Frontiers in neuroinformatics* (2008), p. 11 (cit. on p. 27).

[43] C. Lin et al. «Programming Spiking Neural Networks on Intel's Loihi». In: *Computer* (2018) (cit. on pp. 27, 36, 38, 39).

[44] Daniel Rasmussen. «NengoDL: Combining deep learning and neuromorphic modelling methods». In: *arXiv* (2018) (cit. on p. 27).

[45] François Chollet et al. *Keras*. https://github.com/fchollet/keras. 2015 (cit. on pp. 27, 36).

[46] *CelePixel - Bio-inspired Computer Vision*. 2019. URL: `https://www.celepixel.com/#/Technology` (cit. on p. 28).

[47] *The Dynamic Vision Sensor, iniVation*. 2019. URL: `https://inivation.com/dvs/` (cit. on p. 29).

[48] David Heeger. «Poisson Model of Spike Generation». In: (2000) (cit. on p. 32).

[49] P. U. Diehl, D. Neil, J. Binas, M. Cook, S. Liu, and M. Pfeiffer. «Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing». In: *IJCNN*. 2015 (cit. on p. 36).

[50] Martın Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015 (cit. on p. 37).

[51] Jacques Kaiser et al. «Embodied Event-Driven Random Backpropagation». In: *CoRR* (2019) (cit. on p. 52).

[52] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. «ImageNet: A Large-Scale Hierarchical Image Database». In: *CVPR09* (cit. on p. 61).

[53] Ravi Kanbur. «Pareto's Revenge». In: *Journal of Social and Economic Development* (2005) (cit. on p. 74).