

École polytechnique de Louvain

Embedded Real-Time Inference in Spiking Neural Networks for Neuromorphic IoT Vision Sensors

Author: **Julien VERECKEN**

Supervisors: **David BOL, Christophe DE VLEESCHOUWER**

Readers: **Jean-Didier LEGAT, Martin LEFEBVRE**

Academic year 2019–2020

Master [120] in Electrical Engineering

1	Machine Learning Fundamentals	27
1.1	Introduction	27
1.2	Neural Networks	28
1.2.1	Formulation	28
1.2.2	Optimization	30
1.2.3	The chain rule	32
1.2.4	Back-propagation algorithm	34
1.3	Convolutional Neural Networks	36
1.3.1	Formulation	36
1.3.2	Back-propagation algorithm	38
1.3.3	Convolution variants and down-sampling	40
1.3.4	Advanced models	40
1.3.5	Datasets	42
1.4	Machine learning workflow	42
1.5	Regularization methods	44
1.6	Machine Learning Frameworks	46
1.6.1	Pytorch	46
1.6.2	Training wrappers	47

2	Embedded Machine Learning & Neuromorphic Computing Fundamentals	49
2.1	Challenges in Embedded Machine Learning	49
2.1.1	Context	49
2.1.2	Network structure optimization	50
2.1.3	Exploiting robustness against errors	51
2.1.4	Exploiting sparsity	51
2.1.5	Opportunities	51
2.2	Neuromorphic Computing	52
2.2.1	Dynamic Vision Sensors	52
2.2.2	Event-based datasets	54
2.2.3	Spiking Neural Networks	56
2.2.4	Learning techniques	59
2.2.5	N-CARS state-of-the-art	62
2.2.6	Neuromorphic chips	63
2.2.7	Summary	64

3	Training & analysis of Convolutional Spiking Neural Networks on the N-CARS classification task	65
3.1	Analysis of the N-CARS Dataset	65
3.1.1	Context	65
3.1.2	Dataset Statistics	67
3.2	The Slayer Spiking Neural Network	76
3.2.1	Formulation	76
3.2.2	Slayer back-propagation algorithm	79
3.3	Pytorch framework	85
3.3.1	Slayer Pytorch features	85
3.3.2	Pytorch Lightning Framework	88
3.3.3	Data augmentation	90
3.3.4	Random sampling and experiments	91
3.4	Experiments	91
3.4.1	Baseline model	94
3.4.2	Architecture space exploration	95
3.4.3	Neural parameter space exploration	98
3.4.4	Optimized model	100
3.4.5	Model statistics	102

4	ARM Cortex-A Series Fundamentals	107
4.1	ARM architecture and extensions	107
4.1.1	Introduction	107
4.1.2	Architecture versions	108
4.1.3	Extensions	108
4.1.4	Cortex A-9 processor properties	109
4.2	ARM Processor and Assembly language	110
4.2.1	Processor registers	110
4.2.2	Processor modes and exceptions	111
4.2.3	ARM instruction set	111
4.2.4	Application Binary Interface	115
4.3	VFP and NEON Units	115
4.3.1	Floating-point numbers	115
4.3.2	SIMD instruction principle	117
4.3.3	VFP and NEON engines	117
4.3.4	NEON instruction syntax	120
4.3.5	NEON optimization	124
4.4	Memory systems	127
4.4.1	Motivations	127
4.4.2	Cache architecture	128
4.4.3	Cache policies	131
4.4.4	Cache buffers	131
4.4.5	Cache hierarchy	132
4.4.6	Cache drawbacks	133
4.4.7	Performance measurements	133
4.5	Performance Monitoring Unit	134
4.6	Bare-metal framework	135

4.6.1	DE10-Nano Board features	136
4.6.2	Boot flow	137
4.6.3	SD/MMC partition table	139
4.6.4	The GCC toolchain	139
4.6.5	Linkerscript and startup sequence	142
4.6.6	Application initialization code	143
4.6.7	Project organization	144
4.6.8	Summary	144

5	Optimization of Convolutional Spiking Neural Networks on the Cortex-A9 Processor	146
5.1	Slayer real-time reformulation	147
5.1.1	Infinite impulse response model	148
5.1.2	Event-based dense layer	156
5.1.3	Event-based convolutional layer	157
5.1.4	Target application structure	163
5.2	NEON Optimization	169
5.2.1	Prior optimizations	169
5.2.2	Profiling of the initial solution	172
5.2.3	Update method optimization for 32-bit floating-point	173
5.2.4	NEON Intrinsics performances and enhancements	181
5.2.5	Update method optimization for 16-bit floating-point data-types	189
5.2.6	Layer operations optimization for 32-bit floating-point	193
5.2.7	Layer operations optimization for 16-bit floating-point	198
5.2.8	Performance summary	200
5.3	Model validation of performances and accuracy	201
5.3.1	Barman profiling for the 32-bit floating-point implementation	201
5.3.2	Barman profiling for the 16-bit floating-point implementation	204
5.3.3	Model timing and accuracy	211
5.3.4	Comparison with state-of-the-art real-time solutions	213

Appendices	225
A Pytorch YAML configuration files	226
A.1 Baseline model	226
A.2 Optimized model	227
B Virtual Memory Systems	228
B.1 Virtual address space	228
B.2 Translation tables	228
B.3 Table lookaside buffer	230
C C++ and Assembly methods	232
C.1 Update method	232
C.1.1 32-bit floating-point implementations	232
C.1.2 16-bit floating-point implementation	238
C.2 Dense operation method	241
C.2.1 32-bit floating-point implementation	241
C.2.2 16-bit floating-point implementation	242
C.3 Convolution operation method	244
C.3.1 32-bit floating-point implementation	246
C.3.2 16-bit floating-point implementation	248

LIST OF FIGURES

1.1	MLP Topology.	29
1.2	1D Gradient Descent Algorithm.	31
1.3	General model with two layers.	32
1.4	Back-propagation in a general model with two layers.	33
1.5	Sigmoid and ReLU activation functions.	35
1.6	2D multi-channel convolution with 3×3 filters.	37
1.7	ImageNet challenge state-of-the-art accuracies, reproduced from [5].	41
1.8	VGG-16 Model Architecture [2].	41
1.9	Overfitting example with a polynomial model. Blue dots are the training set, green dots are the validation set. The red model to the left interpolates exactly the blue dots with a tenth order polynomial. Hence, it has a training MSE of 0. The purple model to the right is a regularized third order polynomial with much less parameters ; it displays a higher MSE error on the training but generalizes on the validation set with an error in the same order of magnitude.	43
1.10	Early stopping regularization by monitoring the validation accuracy at each epoch.	45
1.11	Customized function integration in the Pytorch Framework, reproduced from [39]. The new function is derived from Autograd's <i>Function</i> class and requires the implementation of the forward and backward methods. The forward method produces the output of the layer from its input and the backward pass computes the gradient of the output with respect to every input. From this information, Autograd is able to generate the computational graph. This example involves reproducing a simple exponential function.	47

2.1	Comparison between the output of a standard APS and a DVS, adapted from [9]. The output of a DVS cameras only contains pixel that felt variation in brightness. “(a) Sequence of APS output images with an exposure time of 0.3 ms; (b) Short DVS output time slices of the same scene, white and black pixels show the ON and OFF events; (c) Close-up of the arm, ball and racket over 31 ms time slice using the gray level to show the event time.” – Brandli et al.	53
2.2	Example of produced samples for the N-MNIST and N-Caltech datasets, reproduced from [58]. Samples are created from three saccades applied on a DVS camera set on a pan-tilt mount. Original samples are displayed on top and events associated with three saccades are shown below. Samples are constructed by concatenation of these saccades.	55
2.3	Example of samples from the event-based DVS-Gesture dataset along with a frame-based comparison, reproduced from [20]. “Top: 24-bit RGB video frames of the gestures used in this experiment. From left to right the gestures are hand-wave, large-circle, hand-roler, air-guitar, and air-drums. For compactness, only one hand is shown for hand-wave and large-circle, and only one direction for large-circle. Bottom: DVS ‘frames’ created by superimposing spikes over a 5 ms window aligned to the onset of each RGB video frame.” – Amir et al.	56
2.4	Slayer [13] output prediction on a DVS-Gesture Sample. Top left : simultaneous events at a given time-step. Top right : spike count distribution of the output layer (prediction). Bottom : outputs spikes of each neuron of the last layer vs. time. Even if only half of the input events has been processed, the network is already able to output the correct prediction with significant discrimination, based on the accumulated number of spikes.	59
3.1	N-CARS labelling protocol, reproduced from [17]. “(a) Asynchronous gray-scale events are generated by the ATIS camera. (b) By cumulating gray-scale events, we create gray-scale images every 100ms. (c) We run state-of-the-art R-CNNs to automatically detect bounding-boxes contains cars or background samples. (d) The bounding-boxes are manually cleaned to ensure the dataset contains only correctly labeled samples. (e) The spatial bounding-box is converted to a spatio-temporal bounding-box of 100 ms. Each event stream contained in the spatio-temporal bounding-box corresponds to a sample in the N-CARS dataset.” – Sironi et al.	66
3.2	The image show events accumulated over a period of 4 ms, red pixels indicate ON events, blue pixels indicate OFF events and pink pixels indicate a simultaneous ON/OFF event.	67
3.3	Distribution for the width, height and number of events for each class in the 1K subset of N-CARS.	68
3.4	Distributions of the width, height and number of events in the 1K subset of N-CARS.	70

3.5	Correlation between the number of events and the sample's image size (pixels) in the $1K$ subset of N-CARS.	70
3.6	Distributions of the time stamps, polarity and location of the events in the $1K$ subset of N-CARS.	71
3.7	Distributions of the number of events in a sample for a single pixel in the $1K$ subset of N-CARS.	72
3.8	Distributions of the time between events for pixels that output more than one event, computed for samples in the $1K$ subset of N-CARS.	72
3.9	Distributions of the number of events in 1 ms time-bins and the corresponding percentage with respect to the sample image size, for samples in the $1K$ subset of N-CARS.	74
3.10	Sparsity of the samples for 1 ms time bins vs. Time Stamp in the sample, computed for samples in the $1K$ subset of N-CARS. The thick lines represents the average and the shaded area indicates the standard deviation.	75
3.11	Immediate and Refractory kernels	78
3.13	Slayer neuron model.	79
3.12	Slayer neuron behavior. Top : Input spike signal. Middle-top : Membrane potential decomposed into the immediate and refractory response. Middle-bottom : Membrane potential. Bottom : Output spike signal.	80
3.14	Spike escape rate function	83
3.15	Conversion process between the AER format (left) and the dense tensor format (right). The tensor is 4-dimensional, the channel dimension is represented by a color code. Red: 1, Blue, 0.	86
3.16	FIR generation method for the ϵ and ν filters in the Pytorch framework. tSample indicates the total time duration of the sample and Ts indicates the sampling period.	87
3.17	3D Convolution operation with a filter of size $3 \times 3 \times 1$, leaving the third dimension's size unaffected.	88
3.18	Interpolation method comparison on a sample of the MNIST dataset.	90
3.19	Data augmentation example of <i>Car</i> sample of the N-CARS dataset. Refer to Fig. 3.2 for the color annotations.	93
3.20	Training curves for the baseline model with hyper-parameters of Table 3.8 and 3.9. The error metric in percent is measured as Error = 100 - Accuracy [%].	95
3.21	Architecture hyper-parameter design space exploration.	97
3.22	Neuronal hyper-parameter design space exploration.	99

3.23	Training curves for the optimized model with hyper-parameters of Table 3.12, 3.14 and 3.15. The error metric in percent is measured as Error = 100 - Accuracy [%].	101
3.24	Distribution of the weights for each layer of the optimized model.	103
3.25	Distribution of the number of spikes per time-step and per sample class for each layer of the optimized model on the test set of N-CARS.	104
3.26	Distribution of the spike sparsity per time-step and per sample class for each layer of the optimized model on the test set of N-CARS.	104
3.27	Number of events for each layer of the optimized model vs. time-step of the optimized model on the test set of N-CARS. The thick lines represent the average and the shaded area indicates the standard deviation.	105
4.1	Single Precision Floating-point Format	116
4.2	SIMD instruction principle	118
4.3	NEON / VFP register bank reinterpretation	118
4.4	SIMD instruction with single lane operand.	122
4.5	De-interleave behavior upon memory load.	122
4.6	NEON intrinsics array structure for the <code>float32x4_t</code> vector data-type. . . .	125
4.7	Diverging processor and memory performances, reproduced from [97].	127
4.8	Direct mapped cache, adapted from [88].	128
4.9	Example of cache address indexing.	129
4.10	Set-associative cache, adapted from [88]	130
4.11	2-way associative cache illustration, V and D represents the valid and dirty bits. Byte indexing within a word of a cache line is not represented.	130
4.12	Two level memory system with a L1 Harvard cache, reproduced from [88]. . .	132
4.13	Memory map of the Cyclone V MPU, reproduced from [93].	137
4.14	Boot flow of the HPS for a bare-metal application.	137
4.15	SD/MMC partitions and related files.	140
4.16	Code organization in the memory space of the Cyclone V.	143
4.17	Bare-metal Framework for the Cyclone V.	145
5.1	Neuron Update Block Diagram.	149

5.2	IIR Direct Forms I and II, reproduced from [103]	150
5.3	IIR Filter Block Diagram	151
5.4	Basic IIR update rule	152
5.5	FIR and IIR comparison.	153
5.7	FIR update rule	155
5.8	FIR and IIR FLOP Comparison.	156
5.9	Sparse dense layer operation.	157
5.10	Even and odd filter separation in stride 2 convolution.	159
5.11	Overlap of outputs in 1D convolution. Red markers indicate separate events.	161
5.12	Sparse 1D convolution layer operation	161
5.13	Even and odd filter separation in 2D stride 2 convolution.	162
5.14	Sparse 2D convolution layer operation	162
5.15	FLOP of the normal and sparse convolution algorithms.	164
5.16	C++ Spike structure	165
5.17	C++ forward layer operation	166
5.18	Memory repartition in percentage for the optimized model, divided for layers and operations. Absolute values are in Table 5.1.	168
5.19	Layer's input and state memory organization. state[1] and state[2] permute at the end of the update method.	170
5.20	Prior optimization for the IIR update method.	171
5.21	Indexing of a 2D array in memory ; the values arranged in the width dimension are contiguous.	171
5.22	Weight tensor axis ordering for a 5×5 filter.	172
5.23	Execution time distribution per layer for the C implementation.	174
5.24	Execution time distribution of the forward method for the C implementation.	174
5.25	Average execution time repartition for the different layers for the C implementation.	175
5.26	Average execution time repartition for the layer_operation and update methods for the C implementation.	175

5.27	Average execution time repartition for the different layers and methods for the C implementation.	176
5.28	Loop invariants initialization.	178
5.29	Memory de-interleave of states s1 into 2 Q registers.	178
5.30	Loading of the states and filter input into Q registers.	179
5.31	Membrane potential computation and thresholding.	179
5.32	States update and storage.	180
5.33	Update method metrics for the C implementation and different cache configurations and optimization flags.	183
5.34	update method metrics comparison between the C implementation and the NEON intrinsics implementation.	184
5.35	Revised layer memory organization to avoid cache trashing.	186
5.36	Memory alignment hint with the GCC builtin routine.	186
5.37	State and filter input pre-fetching using inline Assembly language.	187
5.38	Update method metrics comparison between the NEON intrinsic implementation and its memory optimized counterpart, for several pre-fetch distances.	188
5.39	Update method output spike sparsity influence. On the left, Thick line : average result over several layer sizes. Shaded area : standard deviation.	189
5.40	update method execution time comparison between all implementations.	190
5.41	update method performance coefficient comparison between all algorithms.	190
5.42	Conversion operations from 16-bit floating-point to 32-bit floating-point data-types for the filter input and state load.	191
5.43	update method metrics comparison between the memory optimized 16-bit and 32-bit floating-point NEON intrinsic implementations.	192
5.44	update method performance coefficient comparison between the memory optimized 16-bit and 32-bit floating-point NEON intrinsic implementations.	193
5.45	Input layer size influence on the dense method execution time C and NEON implementations.	194
5.46	Dense method execution time vs. number of spikes for C and NEON implementations and multiple output channels.	195
5.47	Dense method execution time per spike and per output channel for C and NEON implementations.	196

5.48	Convolution method execution time vs. number of spikes for C and NEON implementations and multiple input and output channels.	196
5.49	Convolution method execution time vs. number of spikes for C and NEON implementations and multiple output channels and input formats.	197
5.50	Convolution method execution time per spike vs. number of output channels for C and NEON implementations and multiple input format.	197
5.51	Convolution method execution time per spike and per output channel for C and NEON implementations and input formats.	198
5.52	Dense method execution time per spike and per output channel for half and single precision data-types.	199
5.53	Convolution method execution time per spike and per output channel for half and single precision data-types.	200
5.54	Execution time distribution per layer for the F32 NEON intrinsic implementation.	202
5.55	Execution time distribution of the forward method for the F32 NEON intrinsic implementation.	202
5.56	Average execution time repartition for the layer_operation and update methods of the F32 NEON intrinsic implementation.	203
5.57	Average execution time time repartition for the different layers and methods for the F32 NEON intrinsic implementation.	203
5.58	Observed characteristics of the layer operation execution time vs. the number of spikes in each layer for the F32 data-type.	205
5.59	Observed performance coefficients during the model's operation for the F32 data-type.	205
5.60	forward method metrics for the F32 implementation for each time-step, averaged over 36 samples. Thick lines indicate the average and shaded areas represent the standard deviation.	206
5.61	Execution time distribution per layer for the F16 NEON intrinsic implementation.	207
5.62	Execution time distribution of the forward method for the F16 NEON intrinsic implementation.	207
5.63	Average execution time repartition for the layer_operation and update methods for the F16 NEON intrinsic implementation.	208
5.64	Average execution time time repartition for the F16 NEON intrinsic implementation.	208
5.65	Observed characteristics of the layer operation execution time vs. the number of spikes in each layer.	209

5.66	Observed performance coefficients during the model's operation.	209
5.67	forward method metrics for the F16 implementation for each time-step, averaged over 36 samples. Thick lines indicate the average and shaded areas represent the standard deviation. (Values at time 0 are removed due to a measurement error.)	210
5.68	Output spike comparison of the Pytorch and C++ F32 implementations for a single sample. Red dots are events present in Pytorch in C++. Blue dots indicate the contrary.	212
5.69	Output spike comparison of the Pytorch and C++ F16 implementations for a single sample. Red dots are events present in Pytorch in C++. Blue dots indicate the contrary	212
A.1	YAML Configuration file for the Baseline Slayer Model.	226
A.2	YAML Configuration file for the Optimized Slayer Model.	227
B.1	Virtual address translation of a 1 MB section descriptor, inspired from [88].	229
B.2	Virtual address translation of a 4 KB page descriptor, inspired from [88].	230
C.1	update method for the C implementation.	232
C.2	update method Assembly code for the C implementation.	233
C.3	update method for the F32 NEON implementation.	235
C.4	update method Assembly code for the F32 NEON implementation.	236
C.5	update method Assembly code for the F32 memory optimized NEON implementation.	237
C.6	update method for the F16 memory optimized NEON implementation.	239
C.7	update method Assembly code for the F16 memory optimized NEON implementation.	241
C.8	Sparse dense_operation method with F32 NEON intrinsics.	241
C.9	Sparse dense_operation method Assembly code for the F32 NEON intrinsic implementation.	242
C.10	Sparse dense_operation method with F16 NEON intrinsics.	243
C.11	Sparse dense_operation method Assembly code for the F16 NEON intrinsic implementation.	244
C.12	Sparse conv_operation method template for any data-type.	245

C.13 Inlined even/Odd indexing method for the sparse convolution.	246
C.14 Inlined accumulate method for the sparse convolution with F32 NEON intrinsics.	246
C.15 Sparse <code>conv_operation</code> method Assembly code for the F32 NEON implemen- tation.	248
C.16 Inlined accumulate method for the sparse convolution with F16 NEON intrinsics.	249
C.17 Sparse <code>conv_operation</code> method Assembly code for the F16 NEON implemen- tation.	251

LIST OF TABLES

2.1	Slayer performance comparison over the MNIST, N-MNIST and DVS-Gesture datasets.	62
2.2	Reported accuracies for the N-CARS dataset.	63
3.1	N-CARS dataset division.	67
3.2	N-CARS sample statistics per class on the full dataset.	68
3.3	N-CARS sample statistics on the full dataset.	69
3.4	Statistics of the number of events and the time difference between events for a single pixel for samples of the N-CARS dataset.	73
3.5	N-CARS event statistics on the $1K$ subset.	73
3.6	Slayer network parameters of the YAML configuration file.	85
3.7	Added parameters in the YAML configuration file for automated model generation.	89
3.8	Baseline model architecture parameters. LR refers to the learning rate of the optimizer.	94
3.9	Baseline model neuron parameters.	94
3.10	Baseline model metrics	95
3.11	Architecture design space exploration hyper-parameter distributions.	95
3.12	Selected architecture hyper-parameters. The convolution layers have filters of size 5×5 and a stride size of 2.	96
3.13	Neuron design space exploration hyper-parameter distributions.	98

3.14	Neuron hyper-parameters of the selected model of the hyper-parameter search.	100
3.15	Scheduler parameters for the optimized model. The initial learning rate LR is multiplied by γ at two specific epochs.	100
3.16	Optimized model metrics.	101
3.17	Reported accuracies for the N-CARS dataset.	102
3.18	Optimized model's statistics of the number of events per time-step, computed on the test set of N-CARS.	106
4.1	ARM data processing instruction format.	112
4.2	ARM memory instruction format for single transfers.	113
4.3	ARM memory instruction format for multiple transfers.	114
4.4	ARM branch instruction format.	114
4.5	IEEE-754 single precision floating point standard, adapted from [88].	116
4.6	NEON and VFP data-types.	119
4.7	NEON processing instruction format.	121
4.8	NEON processing instruction format.	123
4.9	NEON intrinsics vector and array data-type format.	125
4.10	NEON intrinsics function format.	126
4.11	Bare-metal Framework features.	145
5.1	Memory repartition for the optimized model.	167
5.2	Performance coefficients for the optimized methods.	200
5.3	Estimated execution time of the optimized model.	201
5.4	Accuracy and statistics of the execution time per time-step on the N-CARS 100 subset.	211

μ TLB	Micro Translation Lookaside Buffer.
AAPCS	ARM Architecture Procedure Call Standard.
ABI	Application Binary Interface.
AER	Address Event Representation.
AI	Artificial Intelligence.
ALU	Arithmetic Logic Unit.
ANN	Artificial Neural Network.
API	Application Programming Interface.
APS	Active Pixel Sensor.
ATIS	Asynchronous Time-based Image Sensor.
CISC	Complex Instruction Set Computer.
CNN	Convolutional Neural Network.
CPSR	Current Program Status Register.
CPU	Central Processing Unit.
CUDA	Compute Unified Device Architecture.
DDR	Double Data Rate.
DRAM	Dynamic RAM.
DSP	Digital Signal Processor.
DVS	Dynamic Vision Sensor.
EABI	Embedded ABI.
FAT	File Allocation Table.
FIQ	Fast Interrupt Request.
FIR	Finite Impulse Response.
FLOP	Floating-point Operation.
FPGA	Field Programmable Gate Array.
FPSCR	Floating-point Status and Control Register.
FPU	Floating-Point Unit.

GCC	GNU Compiler Collection.
GHRD	Golden Hardware Reference Design.
GPU	Graphics Processing Unit.
HoG	Histogram of Oriented Gradients.
HPS	Hard Processor System.
I/O	Input/Output.
IIR	Infinite Impulse Response.
IoT	Internet of Things.
IRQ	Interrupt Request.
LIF	Leaky Integrate-and-Fire.
LR	Link Register.
LTI	Linear Time-invariant.
MBR	Master Boot Record.
ML	Machine Learning.
MLP	Multi-layer Perceptron.
MMU	Memory Management Unit.
MPU	Multi-Processor Unit.
MSE	Mean Square Error.
OCRAM	On Chip RAM.
OS	Operating System.
PC	Program Counter.
PDF	Probability Density Function.
PE	Processing Element.
PLE	Preload Engine.
PMU	Performance Monitoring Unit.
PSP	Post Synaptic Potential.
R-CNN	Region-based CNN.
RAM	Random Access Memory.
ReLU	Rectified Linear Unit.
RISC	Reduced Instruction Set Computer.
ROM	Read Only Memory.
RTOS	Real-Time OS.
SD/MMC	Secure Digital/Multimedia Card.
SDRAM	Synchronous DRAM.
SGD	Stochastic Gradient Descent.
SIFT	Scale-invariant Feature Transform.
SIMD	Single Instruction Multiple Data.
Slayer	Spike Layer Error Reassignment in Time.
SNN	Spiking Neural Network.
SoC	System on Chip.
SP	Stack Pointer.

SRAM	Static Ram.
SRM	Spike Response Model.
SRS	Spike Response Signal.
SVM	Support Vector Machine.
TF	Transfer Function.
TLB	Translation Lookaside Buffer.
TTBR	Translation Table Base Register.
UART	Universal Asynchronous Receiver-transmitter.
USB	Universal Serial Bus.
VFP	Vector Floating Point.

In recent years, artificial intelligence has attracted a huge interest in image and speech recognition, promising the advent of intelligent technologies in everyday life. Machine learning and computer vision have received a growing interest for solving complex image recognition tasks but often require powerful platforms to operate [1, 2, 3, 4]. One of the next biggest challenges these techniques have to answer is to be integrated inside mobile devices such as smartphones, wearables and other Internet of Things (IoT) nodes [5, 6]. It enables the possibility of local processing of data for real-time applications and avoids relying on cloud computing servers. In order to achieve this goal, it is believed that a paradigm shift towards alternative algorithmic and hardware solutions is necessary in order to lower the energy costs [7]. Inspiration from biological phenomena led to new approaches for treating information called neuromorphic processing. Neuromorphic Event-based cameras [8, 9, 10, 11] mimic the process by which biological retinas send visual information to the brain, with asynchronous events when a change in brightness is detected by a pixel. New machine learning models such as Spiking Neural Networks (SNN) [12, 13, 14] have been developed by taking inspiration from the brain cell in order to process event-based information and achieve low-power, low-latency and brain-inspired learning techniques [7, 15, 16].

The focus in this master thesis is to obtain a real-time embedded low-power implementation of a spiking neural network in an event-based computer vision application. The task is to classify samples of the challenging N-CARS [17] dataset on an embedded low-power platform and achieve minimal latency. The embedded application is developed in software and targeted to the ARM Cortex-A9 processor on a Cyclone V SoC. An accuracy of 88.59 % was obtained on the N-CARS dataset with a convolutional SNN based on the Slayer model [13], competing with state-of-the-art event-based low-power methods (HATS: 90.2 %) [17] and cloud computing approaches (EST: 92.5 %) [18]. A real-time implementation for the optimized model was achieved on the ARM Cortex-A9 processor for 16-bit and 32-bit floating-point data-types, with event-based convolution and dense operations. It demonstrated a 2 ms latency using 4-lane parallel NEON SIMD instructions and optimizes memory accesses with cache pre-fetching. The design is in competition with the HATS [17] FPGA implementation of Sethi et al. [19] with a 3.3 ms latency. This work additionally provides estimations of the execution time of the program in terms of the model parameters and number of events.

ACKNOWLEDGMENTS

My thanks go in particular to my supervisors Professor David Bol and Professor Christophe De Vleeschouwer for their support and the opportunity given to me to conduct this important exercise with considerable latitude.

I am also grateful to Charlotte Frenkel and Martin Lefebvre for their assistance during the year.

In recent years, artificial intelligence (AI) has attracted a huge interest in image and speech recognition, promising the advent of intelligent technologies in everyday life such as self-driving cars and virtual assistants. Machine learning and computer vision have received a growing interest for solving complex image recognition tasks but often require powerful platforms to operate [1, 2, 3, 4]. One of the next biggest challenges these techniques have to answer is to be integrated inside mobile devices such as smartphones, wearables and other Internet of Things (IoT) nodes [5, 6]. It enables the possibility of local processing of data for real-time applications and avoids relying on cloud computing servers. In order to achieve this goal, it is believed that a paradigm shift towards alternative algorithmic and hardware solutions is necessary in order to lower the energy costs [7]. Inspiration from biological phenomena led to new approaches for treating information called neuromorphic processing. In particular, neuromorphic event-based vision is a new expanding field that promises advantages in sensing and processing performances [6].

Event-based cameras [8, 9, 10, 11] mimic the process by which biological retinas send visual information to the brain, with asynchronous events when a change in brightness is detected by a pixel. The event rate adapts depending on the nature of the scene, eliminating redundant information [20] and can therefore give a greater power efficiency compared to conventional frame-based cameras. New machine learning models such as Spiking Neural Networks (SNN) [12, 13, 14] have been developed by taking inspiration from the brain cell in order to process event-based information. Many designs explore different trade-offs with a focus on low-power, low-latency or better learning performances with brain-inspired learning techniques [7, 15, 16].

The focus in this master thesis is to obtain a real-time embedded low-power implementation of a spiking neural network in an event-based computer vision application. The task is to classify samples of the challenging N-CARS [17] dataset on an embedded low-power platform and achieve minimal latency. The embedded application is developed in software and targeted to the ARM Cortex-A9 processor on a Cyclone V SoC.

The achievements of this work are the following

- An accuracy of **88.59 %** was obtained on the N-CARS dataset with the Slayer SNN. An optimized Convolutional Neural Network (CNN) architecture was selected based on an

hyper-parameter design space exploration with data augmentation. This performance competes with state-of-the-art event-based low-power methods (HATS: 90.2 %) [17] and cloud computing approaches (EST: 92.5 %) [18].

- A real-time formulation of the Slayer algorithm has been produced with event-based convolution and dense operations.
- A real-time implementation for the optimized model was achieved on the ARM Cortex-A9 processor and demonstrated a **2 ms latency** for the N-CARS dataset. It leverages 4-lane parallel NEON SIMD instructions for 16-bit and 32-bit floating-point data-types and optimizes memory accesses with cache pre-fetching. The design is in competition with the HATS [17] FPGA implementation of Sethi et al. [19] with a 3.3 ms latency.
- The produced software was profiled in order to derive the execution time of the program in terms of the model parameters and number of events. The performance coefficients could ultimately be used in future works to design models in a multi-objective optimization problem and explore accuracy/latency trade-offs.

This work is structured as follows :

Chapter 1 introduces the basics on supervised machine learning, with an emphasis on CNN. Constitutive equations and training algorithms are detailed as well as interpretations on the architectural choices behind the CNN. A small introduction on deep learning is proposed with a description of current state-of-the-art techniques in computer vision image recognition. The general workflow to train a model on a supervised task is detailed along with the tools used further in this work.

Chapter 2 begins by discussing the challenges regarding the integration of state-of-the-art machine learning models in embedded low-power environments and the large performance gaps that need to be addressed. An overview on optimization techniques that exploits the network architecture, the robustness against errors and the inherent sparsity of deep learning models are presented. In order to further decrease the energy consumption, the neuromorphic computing approach is introduced with a focus on computer vision applications. Event-based camera principles are explained along with their typical performance characteristics. Building on the concepts of neurosciences, the models presented in Chapter 1 are revised and the Spiking Neural Network model is presented. State-of-the-art performances of SNNs are presented for the N-CARS classification task along with the real-time implementations of end-to-end processing chains.

Chapter 3 focuses on training an SNN to solve the N-CARS classification task. Based on its state-of-the-art performances, the Slayer model is selected to accomplish this task and its principle of operation is first analysed in this chapter. Then, a convolutional SNN with the Slayer model is optimized by tuning parameters in automated experiments. This model is used for the rest of this work and optimized to achieve real-time inference on an embedded platform.

Chapter 4 introduces the fundamental elements required to build an application on an embedded system with the ARM Cortex-A9 processor. The objectives of this chapter are therefore to understand the main concepts of the ARM architecture, its memory

systems and SIMD instructions designed for high performance. Finally, a C/C++ Baremetal framework is implemented and establishes the foundation upon which the real-time SNN application is built.

Chapter 5 creates a real-time application for the Slayer model optimized on the N-CARS dataset. It begins with a reformulation of the Slayer model in an event-based algorithm, capable of processing streams of data from an event-based camera. The algorithm is then integrated in the C/C++ Baremetal framework and each method is implemented in order to leverage the features of the ARM Cortex-A9 processor. Several benchmarks are conducted in an iterative approach to extract the performance metrics of the application and further optimize the design.

Chapter 6 summarizes the work of this master thesis and compares the achieved performances with the current state-of-the-art for the N-CARS task. Further work perspectives are then proposed for future improvements.

1.1 Introduction

In recent years, artificial intelligence (AI) has raised an intense hype around its capabilities in image and speech recognition, promising the advent of intelligent technologies in everyday life such as self-driving cars and virtual assistants. This section ought to introduce the basic concepts around *machine learning* (ML) with a particular focus on vision applications. It is based on the excellent textbook of F. Chollet [21] and the machine learning course of W. Hamilton from McGill University [22]. Machine learning differentiates itself from the broader field of AI by changing the paradigm by which a task is solved by an algorithm. Classical programming of automated tasks involves defining a set of rules that apply on data in order to produce desired results. A machine learning system redefines this framework by having the algorithm output a set of rules based on input data and associated desired results. We say that an ML algorithm is trained on data in order to find by itself useful representations to solve the task. In definitive, the goal of an ML algorithm is to be able to generalize the learned patterns to new data for a given application. In this framework, the following denominations are generally used to address the fundamental components of machine learning :

- *Input data* or *samples* are instances of the available data that are fed to the algorithm. They are drawn from a larger collection of samples called a *dataset*.
- *Output predictions* or *inferences* are produced by the algorithm based on its training on the dataset, possibly using *targets* indicating the desired prediction for a given sample.
- A *model* generally refers to a specific machine learning algorithm, designed to solve a task in a particular application.

In general, ML algorithms can be classified into 4 categories :

1. *Supervised learning* is the most common category of algorithms. They consist in

mapping input data with known targets given a dataset that was annotated by humans. These algorithms come in different types :

- Classification algorithms map input data to a target that belong to a set of classes. For example, character recognition algorithms fall into this category.
 - Regression algorithms map input data to real number target. For example, an algorithm that predicts the compressive strength of concrete based on chemical and process parameters solves a regression task.
2. *Unsupervised learning* consists in developing transformations of input data that produces useful representations, without the help of any target. It allows to acquire a better knowledge of the data by exposing relations between input features. Dimensionality reduction and clustering are examples of such algorithms.
 3. *Self-supervised learning* is a subcategory of supervised learning where the targets are automatically generated from the input data. *Auto-encoders* are an example of self-supervised learning where the model finds an encoding and decoding of input data to represent it in a compressed form. The targets are therefore the same as the input data. Weather forecasting algorithms are another instance of such learning process where an atmospheric parameter is predicted based on past measurements.
 4. *Reinforcement learning* is a separate field of machine learning where models learn to perform actions based on information from their environment in order to maximize a reward. Algorithms that learn to play games are an example of such algorithms.

This work focuses on a specific category of models in supervised learning : neural networks. The general architecture of neural networks is presented in the next section along with essential concepts about the learning procedure of this model. Then, a variant of this model is presented : convolutional neural networks.

1.2 Neural Networks

1.2.1 Formulation

A neural network is a ML model inspired by the topology of the brain. Its invention is traced back to 1958 and is still used today in AI applications [23]. Equivalently, the denominations of Multi-layer Perceptron (MLP) and Artificial Neural Network (ANN) refer to this model, presented in the following paragraphs.

A neural network consists in stacks of layers, each containing several *neurons*. Each neuron of a layer (l) is linked with every neuron of the next layer ($l + 1$) via a vector of weights. Due to this structure, such layer is often referred to as *fully-connected* or *dense*. The constitutive equations of the MLP are written in Eq. 1.1.

$$\begin{aligned} \mathbf{z}^{(l+1)} &= \mathbf{x}^{(l)} \cdot \mathbf{W}^{(l)} + \mathbf{b}^{(l)} \\ \mathbf{x}^{(l+1)} &= \phi(\mathbf{z}^{(l+1)}) \end{aligned} \tag{1.1}$$

- $\mathbf{x}^{(l)}$ is the input vector of layer (l) , it has a length $N^{(l)}$.
- $\mathbf{W}^{(l)}$ is the matrix of weights, it has a size $N^{(l)} \times N^{(l+1)}$.
- $\mathbf{b}^{(l)}$ is a vectors of biases, it has a length $N^{(l+1)}$.
- $\mathbf{x}^{(l+1)}$ is the output vector of layer (l) , it has a length $N^{(l+1)}$

The neurons of layer $(l + 1)$ are therefore linked with the neurons of the previous layer via a dot product with a vector of weights. At the layer level, this is analogous to a matrix product. Each neuron therefore receives as input a linear combination between the *activations* of the previous layer and the weights and outputs an activation for the next layer via a non-linear *activation function* ϕ . As a result, elements of $\mathbf{x}$ are often referred as *activations*. This topology is illustrated by the diagram of Fig. 1.1.

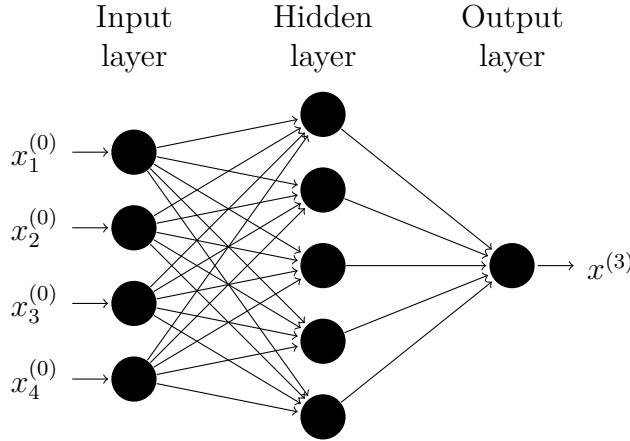


Figure 1.1: MLP Topology.

The input layer accepts vectorized samples from a dataset, the *hidden* layers processes the sample and extracts abstract representations in the data and the output layer produces the desired information. In the case of a classification task, the output layer can output the probability that the input belongs to a given class. For regression tasks, the layer simply returns an unbounded real value.

MLP models have been used for years to solve pattern recognition tasks. In the early decades, application specific methods were used to extract features from the data and feed them into a neural network. For example, in image recognition applications, these methods were designed by hand to extract features such as edges, corners and histogram of gradients (HoG [24], SIFT [25]). This step was fundamental as only small neural networks with a limited amount of layers could be trained, due to training issues and a lack of computing power. This led to a consequent stall in research between the 70s and 90s, otherwise known as the *AI winter*. Nevertheless, with innovations in training techniques [26] and the democratization of computing power via computer servers and Graphical Processing Units (GPU), the research field regained interest in the 90s. Larger neural networks could now be trained on bigger datasets which resulted in a considerable increase in their modeling capabilities. Such *deeper* structures allow to completely integrate the role of the feature extractor and optimize it to build the best representation of the data during its training. Neural networks that processes raw input data are therefore referred to as *deep learning* models [27] : their first layers

extract low-level features that are hierarchically combined in the next layers to produce an output prediction. The increase in performance between hand-crafted feature models and deep learning models was spectacular, especially in the *ImageNet challenge* [28] and marked a new era machine learning. The process by which these networks are trained is described in the following section.

1.2.2 Optimization

When creating the model, the initial weight matrices are filled with random values and there are no reasons to expect that the output of the model can produce a meaningful result. The *training* process therefore involves adjusting gradually the weight values in order to optimize a given metric involving the model outputs and the targets. It takes place in the *training loop*, composed of the following steps :

1. At the beginning of each iteration of the loop (also called *epoch*), a *batch* of input samples and corresponding targets is fetched from the dataset.
2. The model is fed with the input batch and produces a batch of predictions. This is called the *forward pass*.
3. Predictions are compared with the targets using a specific function that measures the mismatch between the two, called the *loss function*. The model's *error* or *loss* also refers to the output value of the loss function for a given batch of input data.
4. The weights of the network are updated based on the loss value in a phase called the *backward pass*.

The goal of the learning process in a classification task is to maximize a given metric on the training dataset. The choice of the right metric depends on the specificities of the task to be solved.

1. *Accuracy* is in general an adequate choice when dealing with a dataset of nearly equiprobable classes.
2. *Precision* and *recall* metrics are sometimes preferred in situations where a pronounced class imbalance is present in the dataset. It could also be used when the impact of a *false-positive* or *false-negative* prediction is fundamentally different, in a medical application for example.

In a binary classification task, the metrics are computed as :

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}} \quad (1.2)$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (1.3)$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (1.4)$$

where TP and TN indicates true positives and negatives while FP and FN indicates false positives and negatives.

The problem is that those metrics are discrete in nature : the output prediction is compared with the target in a binary correct/incorrect manner. On the other hand, a loss function provides a continuous range value when measuring the mismatch between the predictions and the targets. This loss therefore needs to be minimized during the training phase. However, the loss values fail to give any significant interpretation compared to accuracy, precision and recall metrics. A commonly used loss function for classifications is the *cross-entropy loss* function ; it relates to the negative log-likelihood of the target data, given the model's input batch and weights. Regression models usually choose the *mean square error* between the model predictions and the targets.

An optimal way to update the weights based on the computation of the loss is to "move" them in a direction that decreases the loss value. It therefore involves computing the gradient of the loss with respect to the weights of the model and moving the weights in a direction that is opposite to the gradient. This fundamental process can be written as

$$w_{\text{new}} = w_{\text{old}} - \alpha \cdot \left. \frac{\partial E}{\partial w} \right|_{\mathbf{x}} \quad (1.5)$$

where w refers to a weight of the model, the derivative of the loss E with respect to w is evaluated on the input batch $\mathbf{x}$ and α is a scalar called the *learning rate*. This process is repeated for each epoch and the weights are progressively moved towards a minimum of the loss function. This is illustrated in Fig. 1.2

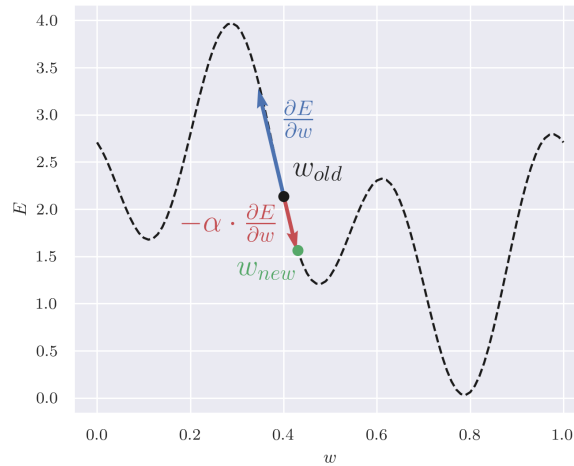


Figure 1.2: 1D Gradient Descent Algorithm.

This method is called the *batch stochastic gradient descent* (SGD) algorithm, as the input batch is drawn at random from the dataset. Variants of this algorithm (Adam, Adagrad, RMSProp [29]), are preferred in practice when training a neural network, they take into account additional information such as the previous weight updates and higher order derivatives to enhance the weight update rule. The main additions are the following :

1. Based on past weight updates, a *momentum* is added in the weight update rule. It allows to gain a better convergence speed and escape local minima of the loss function.

2. Based on higher order derivatives, the learning rate is adapted for better convergence properties.

These algorithms are generally referred under the term *optimizer*. The computation of the gradient of the loss with respect to the weights of the model is an involved analytical process. As the model can be considered as large analytical function where each weight can be viewed as an intermediate variable, the derivation of the gradient involves the *chain rule*. This rule is reminded in the next section for general machine learning models and introduces notations used for the rest of this work.

1.2.3 The chain rule

Let us consider a machine learning model composed of 2 layers with transformations A and B respectively. They are each parametrized with vectors of weights $\mathbf{w}_A$ and $\mathbf{w}_B$. For the sake of generality, let us consider that each layer outputs a vector with a different size. Finally, the output of the last layer enters a loss function that returns a scalar metric. We therefore have the following equations :

$\mathbf{x}$ is the input with size N_x .

$\mathbf{y} = A(\mathbf{x}, \mathbf{w}_A)$ is the output of the first layer with size N_y .

$\mathbf{z} = B(\mathbf{y}, \mathbf{w}_B)$ is the output of the second layer with size N_z .

$E = L(\mathbf{z}, \hat{\mathbf{z}})$ is the error metric, $\hat{\mathbf{z}}$ is the desired output.

The model is illustrated in Fig. 1.3 for clarity.

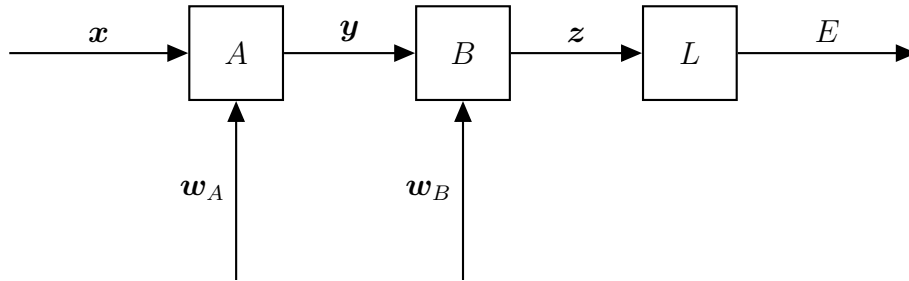


Figure 1.3: General model with two layers.

The gradient descent algorithm adapts the weight of the model based on the gradient of the loss function with respect to the weights of the model. The quantities that we wish to compute are therefore

$$\frac{\partial E}{\partial \mathbf{w}_A} \quad \text{and} \quad \frac{\partial E}{\partial \mathbf{w}_B}.$$

These quantities can be computed from the output of the last layer via the *chain rule*. Proceeding from the last layer back to the first for the chain rule substitution is also known as the *back-propagation* algorithm. In the subsequent developments the notation of Eq. 1.6 is used.

$$da := \frac{\partial E}{\partial a} \quad \text{with } a \text{ a variable of the model.} \quad (1.6)$$

Starting at the output of the model, the loss function L is known, therefore the gradient is

$$dz = \frac{\partial E}{\partial z}. \quad (1.7)$$

The gradient of the loss with respect to the weights of the second layer is computed using the chain rule.

$$\frac{\partial E}{\partial w_{A,i}} = \sum_{k=0}^{N_z-1} \frac{\partial E}{\partial z_k} \frac{\partial z_k}{\partial w_{A,i}}. \quad (1.8)$$

The partial derivatives depend on the transformation B . This equation may be represented with the following matrix product including the quantity dz .

$$\frac{\partial E}{\partial w_A} = dz \cdot \underbrace{\frac{\partial z}{\partial w_A}}_{J_{w_A}} \quad (1.9)$$

Computing the gradient with respect to the weights therefore systematically involves computing a matrix product of the output gradient with the *Jacobian matrix* with entries $J_{w_A,i,j} = \frac{\partial z_i}{\partial w_{A,j}}$. Similarly, the gradient of the loss with respect to the weights of the first layer is calculated as

$$\frac{\partial E}{\partial w_B} = dy \cdot \underbrace{\frac{\partial y}{\partial w_B}}_{J_{w_B}}. \quad (1.10)$$

The only problem here is that the quantity dy has not yet been computed. It is easily done using the second Jacobian matrix of the transformation in Eq. 1.11.

$$dy = dz \cdot \underbrace{\frac{\partial z}{\partial y}}_{J_y} \quad (1.11)$$

It should now become apparent that the back-propagation algorithm requires that a layer computes the gradient of the loss with respect to its weights locally and then "back-propagates" the gradient with respect to the inputs to the previous layer in order for it to achieve the same operation. This operation can be automated, as explained in Section 1.6.1. The back-propagation for the general model of Fig. 1.3 is illustrated in Fig. 1.4. It should be mentioned at this point that the back-propagation is only feasible in models that show an *acyclic graph* structure. In other terms, the model does not contain loops.

The next section focuses on the derivation of the back-propagation algorithm for the MLP.

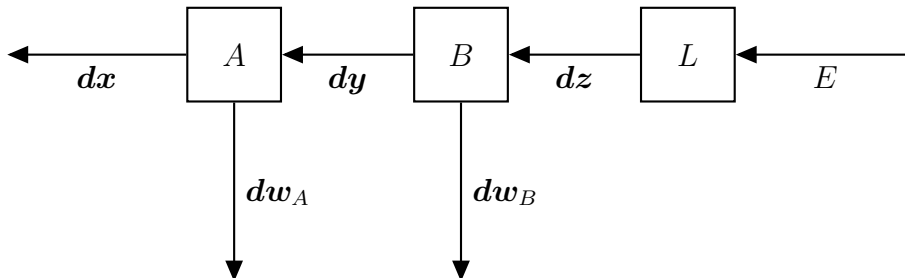


Figure 1.4: Back-propagation in a general model with two layers.

1.2.4 Back-propagation algorithm

The back-propagation derivation assumes that a loss function L has been chosen and that its gradient with respect to the output of the last layer (l_f) of the network is known. Building on the concepts of the last section, the back-propagation rule is derived by recursion. The computations for the propagation from one layer to the previous suffice to build the back-propagation for the entire network. The recursion starts with the gradient of the error at the end of the network : $\frac{\partial E}{\partial \mathbf{x}^{(l_f)}}$.

Given the gradient of the error with respect to the output of layer $(l + 1)$, the following relations are derived.

The output gradient is given either by the loss gradient or the back-propagation of the next layer.

$$\mathbf{dx}^{(l+1)} = \frac{\partial E}{\partial \mathbf{x}^{(l+1)}} \quad (1.12)$$

An intermediate result is the gradient of the loss with respect to $\mathbf{z}^{(l+1)}$ and it is given by

$$\begin{aligned} dz_i^{(l+1)} &= \sum_{k=0}^{N^{(l+1)}-1} dx_k^{(l+1)} \frac{\partial x_k^{(l+1)}}{\partial z_i^{(l+1)}} \\ \frac{\partial x_k^{(l+1)}}{\partial z_i^{(l+1)}} &= \begin{cases} \phi'(z_i^{(l+1)}) & \text{if } k = i \\ 0 & \text{otherwise} \end{cases} \\ \mathbf{dz}^{(l+1)} &= \mathbf{dx}^{(l+1)} \cdot \text{diag}\{\phi'(\mathbf{z}^{(l+1)})\} \end{aligned} \quad (1.13)$$

The input gradient to propagate to the previous layer is therefore given by

$$\begin{aligned} dx_i^{(l)} &= \sum_{k=0}^{N^{(l+1)}-1} dz_k^{(l+1)} \frac{\partial z_k^{(l+1)}}{\partial x_i^{(l)}} \\ &= \sum_{k=0}^{N^{(l+1)}-1} dz_k^{(l+1)} \left(\sum_{j=0}^{N^{(l)}-1} \frac{\partial x_j^{(l)}}{\partial x_i^{(l)}} W_{j,k}^{(l)} \right) \\ &= \sum_{k=0}^{N^{(l+1)}-1} dz_k^{(l+1)} W_{i,k}^{(l)} \\ \mathbf{dx}^{(l)} &= \mathbf{dz}^{(l+1)} \cdot (\mathbf{W}^{(l)})^T \end{aligned} \quad (1.14)$$

The weight gradient for the update rule is given by

$$\begin{aligned} dW_{i,j}^{(l)} &= \sum_{k=0}^{N^{(l+1)}-1} dz_k^{(l+1)} \frac{\partial z_k^{(l+1)}}{\partial W_{i,j}^{(l)}} \\ &= \sum_{k=0}^{N^{(l+1)}-1} dz_k^{(l+1)} \left(\sum_{m=0}^{N^{(l)}-1} x_m^{(l)} \frac{\partial W_{m,k}^{(l)}}{\partial W_{i,j}^{(l)}} \right) \\ &= dz_j^{(l+1)} x_i^{(l)} \\ \mathbf{dW}^{(l)} &= (\mathbf{x}^{(l)})^T \cdot \mathbf{dz}^{(l+1)} \end{aligned} \quad (1.15)$$

The bias gradient for the update rule is given by

$$\begin{aligned}
db_i^{(l)} &= \sum_{k=0}^{N^{(l+1)}-1} dz_k^{(l+1)} \frac{\partial z_k^{(l+1)}}{\partial b_i^{(l)}} \\
&= \sum_{k=0}^{N^{(l+1)}-1} dz_k^{(l+1)} \frac{\partial b_k^{(l)}}{\partial b_i^{(l)}} \\
&= dz_i^{(l+1)} \\
\mathbf{db}^{(l)} &= \mathbf{dz}^{(l+1)}
\end{aligned} \tag{1.17}$$

As it can be seen in these derivations, the gradient of the activation function must be defined and must display stable properties for efficient training. The activation function may be selected to reach optimal performances on the task to solve. Historically, the sigmoid function [21] was selected as a smooth differentiable function and exhibited an interesting thresholding behavior that was assimilated to binary decisions taken in the network. However, this choice was questioned after the advent of deeper layer structures. Indeed, the *vanishing gradient* [30] is a problem that arises when training the network and involves gradients of the activation function that are too small to perform the weight update in deep layers. To solve this problem, the Rectified Linear Unit (ReLU) function was introduced and displayed better sub-gradient properties (as the function is not differentiable in the strict sense, an alternative definition of the gradient is used). The sigmoid and ReLU function formulas are displayed in Eq. 1.18 and Eq. 1.19 respectively and illustrated in Fig. 1.5.

$$\phi_{\text{sigmoid}}(x) = \frac{1}{1 + \exp(-x)} \tag{1.18}$$

$$\phi_{\text{ReLU}}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{1.19}$$

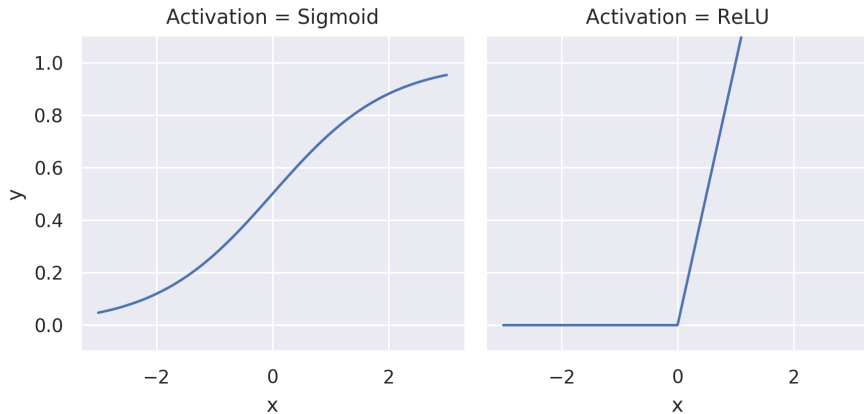


Figure 1.5: Sigmoid and ReLU activation functions.

1.3 Convolutional Neural Networks

1.3.1 Formulation

The fundamental problems behind the classic neural network are twofold:

1. All neurons in the layer (l) are connected with all neurons of the next layer ($l + 1$). The weight matrix therefore has a size $(N^{(l)} + 1) \cdot N^{(l+1)}$ which can grow extremely fast when dealing with high dimensional data, increasing the complexity of the model. This structure is therefore unpractical and nearly impossible to train for the large amounts of neurons, typically required to process pixels of image data.
2. Dense layers learn global patterns because of their connection, this can be a problem when a pattern is expected to be localized in the input features. This is especially the case when dealing with image data as patterns can be found in localized 2D regions.

The Convolutional Neural Network (CNN) solves these problems by proposing a new architecture inspired by visual neurosciences based on the successive filtering of the input image. The model therefore processes 3D tensors. The two first dimensions of these tensors are *spatial* dimensions and the third is the *depth* dimension, also called the *channel* dimension. A convolution operation takes place in the spatial dimensions with a bank of several 3D filters. Each filter has a width K_w and height K_h and a channel size equal to the one of the layer input. In this configuration, each channel of the input is convolved with the corresponding filter's channel, the produced 3D tensor is then summed on the channel dimension to produce a single output channel. The output of the convolutional layer retains a 3D format and is called a *feature map*. Several output channels are produced using several 3D filters, as illustrated in Fig. 1.6. The filter bank, or weight tensor, is therefore a 4D tensor. At last, the bias is common to all elements of the output feature map. The constitutive equations of the CNN are written in Eq. 1.20.

$$\begin{aligned}
 \mathbf{Z}_{c_o}^{(l+1)} &= \sum_{c_i=0}^{C^{(l)}-1} \left(\mathbf{X}_{c_i}^{(l)} \otimes \mathbf{W}_{c_o, c_i}^{(l)} \right) + \mathbf{1}^T b_{c_o}^{(l)} \mathbf{1} \\
 Z_{c_o, h_o, w_o}^{(l+1)} &= \sum_{c_i=0}^{C^{(l)}-1} \sum_{k_h=0}^{K_h^{(l)}-1} \sum_{k_w=0}^{K_w^{(l)}-1} \left(X_{c_i, h_o-k_h, w_o-k_w}^{(l)} W_{c_o, c_i, k_h, k_w}^{(l)} \right) + b_{c_o}^{(l)} \\
 \mathbf{X}_{c_o}^{(l+1)} &= \phi(\mathbf{Z}_{c_o}^{(l+1)})
 \end{aligned} \tag{1.20}$$

- $\mathbf{X}^{(l)}$ is the 3D input tensor of layer (l), it has a size $C^{(l)} \times N_h^{(l)} \times N_w^{(l)}$.
- $\mathbf{W}^{(l)}$ is the 4D weight tensor, it has a size $C^{(l+1)} \times C^{(l)} \times K_h^{(l)} \times K_w^{(l)}$.
- $\mathbf{b}^{(l)}$ is the vector of biases, it has a length $C^{(l+1)}$.
- $\mathbf{X}^{(l+1)}$ is the 3D output tensor of layer (l), it has a size $C^{(l+1)} \times N_h^{(l+1)} \times N_w^{(l+1)}$

The following notation is therefore used for the indices

- c_i spans the input channel dimension of size $C^{(l)}$.
- h_i spans the input height dimension of size $N_h^{(l)}$.
- w_i spans the input width dimension of size $N_w^{(l)}$.

The suffix o instead of i is used when referring to the output layer dimensions.

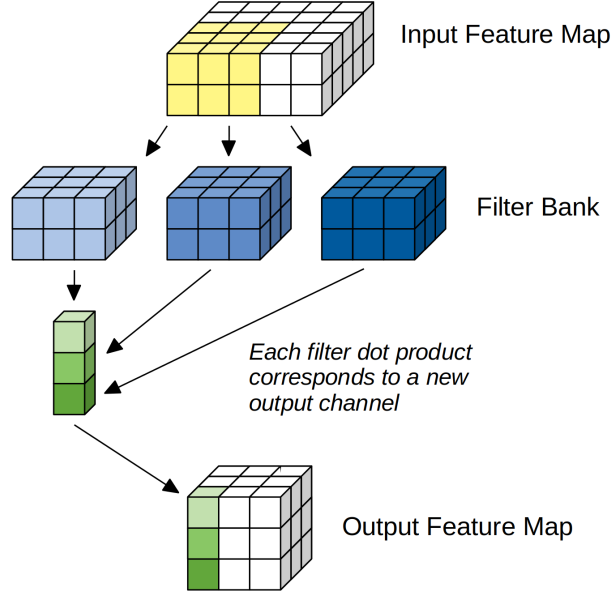


Figure 1.6: 2D multi-channel convolution with 3×3 filters.

A CNN therefore display two key characteristics thanks to the convolution operations :

1. The learned patterns are *translation-invariant* since the filter is a small patch that is swept across the input image. A pattern learned in one location of the input image can be detected and reinforced when located at a different place in another image. In contrast with dense layers, a CNN learns *localized* patterns. It further exploits the 2D topology of images thanks to the 2D convolution operations. The number of parameters is greatly reduced as they are shared between the neurons of a layer.
2. *Spatial hierarchies* of patterns can be learned with multiple layers of convolutions, with an increasing complexity. Because of the limited number of parameters inside a filter, the first layers will focus on learning simplistic patterns such as edges and corners in the input image. The next layers will receive this information and build larger patterns to detect more abstract representations. As objects in the external world have the same hierarchical property, the CNN acts as a powerful feature extractor for image data.

State-of-the-art models on image datasets rely on several convolutional layers in order to extract visual features. The obtained features are fed to a series of dense layers to perform the classification. Performance and state-of-the-art models are discussed in Section 1.3.4.

1.3.2 Back-propagation algorithm

Similar to what was done in Section 1.2.4, the back-propagation algorithm is derived for CNNs. Because it is substantially more complex due to the multiple indices in the notation, we will first derive the back-propagation in a one-dimensional convolution with a single channel. This exercise is useful since this result is re-used later in Section 3.2.2. The operation we consider here is therefore given in Eq. 1.21.

$$\begin{aligned}\mathbf{z}^{(l+1)} &= \left(\mathbf{x}^{(l)} \circledast \mathbf{w}^{(l)}\right) + b^{(l)} \mathbf{1}^T \\ z_i^{(l+1)} &= \sum_{k=0}^{K^{(l)}-1} \left(x_{i-k}^{(l)} w_k^{(l)}\right) + b^{(l)}\end{aligned}\tag{1.21}$$

- $\mathbf{x}^{(l)}$ is the input vector of layer (l) , it has a length $N^{(l)}$.
- $\mathbf{w}^{(l)}$ is the filter of layer (l) , it has a length $K^{(l)}$.
- $b^{(l)}$ is the unique bias.
- $\mathbf{x}^{(l+1)}$ is the input vector of layer $(l+1)$, it has a length $N^{(l+1)}$

As explained later in Section 1.3.3, the convolution operator is the *full* convolution, therefore the output length equals

$$N^{(l+1)} = N^{(l)} + K^{(l)} - 1.$$

The developments do not include the non-linear function ϕ since it has the same influence as in the dense layers. The recursion starts with the gradient of the error at the end of the network, denoted as $\frac{\partial E}{\partial \mathbf{x}^{(l_F)}}$.

Given the gradient of the error with respect to the output of layer $(l+1)$, the following relations are derived.

The gradient of the loss with respect to $z^{(l+1)}$ is given by

$$d\mathbf{z}^{(l+1)} = d\mathbf{x}^{(l+1)} \cdot \text{diag}\{\phi'(\mathbf{z}^{(l+1)})\}\tag{1.22}$$

The input gradient is therefore given by

$$\begin{aligned}dx_i^{(l)} &= \sum_{k=0}^{N^{(l+1)}-1} dz_k^{(l+1)} \frac{\partial z_k^{(l+1)}}{\partial x_i^{(l)}} \\ &= \sum_{k=0}^{N^{(l+1)}-1} dz_k^{(l+1)} \left(\sum_{j=0}^{K^{(l)}-1} \frac{\partial x_{k-j}^{(l)}}{\partial x_i^{(l)}} w_j^{(l)} \right) \\ &= \sum_{k=0}^{N^{(l+1)}-1} dz_k^{(l+1)} w_{k-i}^{(l)}\end{aligned}\tag{1.23}$$

The result is the cross-correlation between the gradient of $z^{(l+1)}$ and the filter.

$$\mathbf{d}\mathbf{x}^{(l)} = \mathbf{d}\mathbf{z}^{(l+1)} \otimes \mathbf{w}^{(l)} \quad (1.24)$$

The weight gradient is given by

$$\begin{aligned} dw_i^{(l)} &= \sum_{k=0}^{N^{(l+1)}-1} dz_k^{(l+1)} \frac{\partial z_k^{(l+1)}}{\partial w_{i,j}^{(l)}} \\ &= \sum_{k=0}^{N^{(l+1)}-1} dz_k^{(l+1)} \left(\sum_{j=0}^{K^{(l)}-1} x_{k-j}^{(l)} \frac{\partial w_j^{(l)}}{\partial w_i^{(l)}} \right) \\ &= \sum_{k=0}^{N^{(l+1)}-1} dz_k^{(l+1)} x_{k-i}^{(l)} \\ \mathbf{d}\mathbf{w}^{(l)} &= \mathbf{d}\mathbf{z}^{(l+1)} \otimes \mathbf{x}^{(l+1)} \end{aligned} \quad (1.25)$$

The bias gradient is given by

$$\begin{aligned} db^{(l)} &= \sum_{k=0}^{N^{(l+1)}-1} dz_k^{(l+1)} \frac{\partial z_k^{(l+1)}}{\partial b^{(l)}} \\ &= \sum_{k=0}^{N^{(l+1)}-1} dz_k^{(l+1)} \\ db^{(l)} &= \mathbf{d}\mathbf{z}^{(l+1)} \cdot \mathbf{1}^T \end{aligned} \quad (1.26)$$

Thanks to these simplifications, the derivations for the back-propagation of actual CNNs are now intuitive. Recalling Eq. 1.20 and given the gradient of the error with respect to the output of layer $(l+1)$, the following relations are derived.

The output gradient is given either by the loss gradient or the back-propagation of the next layer.

$$\mathbf{d}\mathbf{X}^{(l+1)} = \frac{\partial E}{\partial \mathbf{X}^{(l+1)}} \quad (1.27)$$

The gradient of the loss with respect to $Z^{(l+1)}$ is given by

$$\mathbf{d}\mathbf{Z}^{(l+1)} = \mathbf{d}\mathbf{X}^{(l+1)} \cdot \text{diag}\{\phi'(\mathbf{Z}^{(l+1)})\} \quad (1.28)$$

The input gradient is given by

$$\mathbf{d}\mathbf{X}_{c_i}^{(l)} = \sum_{c_o=0}^{C^{(l+1)}-1} \mathbf{d}\mathbf{Z}_{c_o}^{(l+1)} \otimes \mathbf{W}_{c_o, c_i}^{(l)} \quad (1.30)$$

The weight gradient is given by

$$\mathbf{d}\mathbf{W}_{c_o, c_i}^{(l)} = \mathbf{d}\mathbf{Z}_{c_o}^{(l+1)} \otimes \mathbf{X}_{c_i}^{(l)} \quad (1.31)$$

The bias gradient is given by

$$db_{c_o}^{(l)} = \mathbf{1} \cdot \mathbf{d}\mathbf{Z}_{c_o}^{(l+1)} \cdot \mathbf{1}^T \quad (1.32)$$

1.3.3 Convolution variants and down-sampling

A convolution operation essentially reduces the height and width of an image since the filter has a spatial size of its own. This convolution operation is referred in the literature as the *valid convolution*. In order to avoid this spatial shrinking effect, the input image can be *padded* with zeros on the edges in order to conserve the same input format. Another parameter that can influence the output size of the feature maps is the *stride*. If the distance between two successive applications of the filter patch on the input is larger than one, the convolution is said to be *strided*. It is equivalent to a down-sampling of the output feature maps by the stride factor, as shown in Eq. 1.33.

$$Z_{c_o, h_o, w_o}^{(l+1)} = \sum_{c_i=0}^{C^{(l)}-1} \sum_{k_h=0}^{K_h^{(l)}-1} \sum_{k_w=0}^{K_w^{(l)}-1} \left(X_{c_i, s \cdot h_o - k_h, s \cdot w_o - k_w}^{(l)} W_{c_o, c_i, k_h, k_w}^{(l)} \right) + b_{c_o}^{(l)} \quad (1.33)$$

where s is the stride factor. The output size of the convolution is therefore given by

$$N^{(l+1)} = \frac{(N^{(l)} + p - K)}{s} + 1 \quad (1.34)$$

with p the padding length.

Down-sampling is effectively used to drastically reduce the number of neurons between convolutional layers in order to lower their complexity and induce the learning of spatial hierarchies [21]. Indeed, a down-sampling of the feature map allows the next convolutional layer to learn patterns on a larger scale. Once the number of neurons is sufficiently low, dense layers are used as the last layers to perform the classification or regression task. Another popular technique for down-sampling is the *max-pooling* method. The feature map is divided into local patches, usually of size 2×2 and the maximum activation is selected.

1.3.4 Advanced models

Advanced models involving convolutional layers are most often benchmarked on the ImageNet dataset [28]. It regroups around 1.3 million of colour images belonging to one thousand different object categories. The performance comparison in Fig. 1.7 displays the evolution of the error metric in the last recent years, for different models.

Among them, AlexNet [1] drastically outperformed its previous competitors and marked the beginning of the deep learning era in image classification. In 2014, VGG-16 [2] beat AlexNet with a large architecture displayed in Fig. 1.8. It contains 16 convolutional layers of 3×3 filters that are interleaved with several pooling layers to down-sample the feature maps. The last layer contains 4096 neurons and is connected to a stack of dense layers to perform the classification. In total, the network contains around 138 million parameters. The convolutional part of this model, trained on the ImageNet dataset is often used in other applications as a powerful feature extractor. Since the model was trained on a large panel of everyday objects present in the dataset, the learned representations may be used to solve other image classification tasks. Indeed, with *transfer learning*, the dense layers responsible for the classification are changed and trained from scratch while the weights of VGG-16 are downloaded and frozen during the training phase. In addition, a fine-tuning of

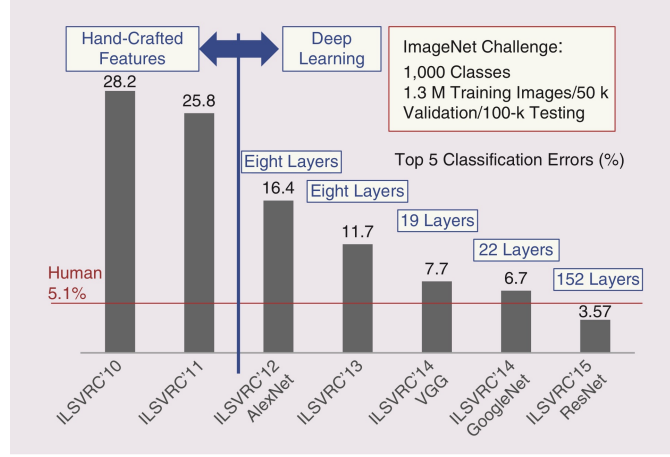


Figure 1.7: ImageNet challenge state-of-the-art accuracies, reproduced from [5].

the weights of CNN feature extractor can be conducted with small learning rates to achieve best performances. This technique has become a goto solution when implementing an image classification model thanks to the publicly available optimized models on ImageNet. It requires however powerful computing resources due to the complexity of such models.

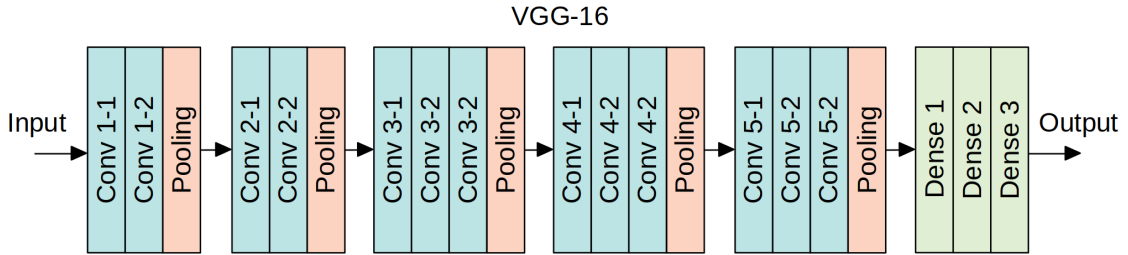


Figure 1.8: VGG-16 Model Architecture [2].

More advanced CNN classifiers include GoogLeNet [3] which introduces *inception* blocks that combines multi-scale convolutions and ResNet [4] that comprises connections skipping layers, alleviating the vanishing-gradient problem in deep network architectures.

As the ImageNet challenge is considered to be a solved task, contemporary research focuses on advanced subjects, still involving the use of CNNs. Among them, region-level object detection tries to predict a bounding box around detected objects in an image. Techniques vary from predicting large amounts of region proposals and classify each one of them as in Region-based CNNs (R-CNNs), using a conventional CNN backbone with a Region of Interest layer or a region proposal network as in Fast R-CNN and Faster R-CNN [31], or predicting simultaneously classes, position and widths of boxes with a regression formulation as in YOLO [32]. In addition, image segmentation models apply different techniques to different objects in a region in a pixel-wise manner instead of boxes. Fully Convolutional Networks (FCN) allow to up-sample a fully connected representation back to an image and achieve classification at the pixel level. Mask R-CNN is currently the most advanced model using a backbone ResNet for classification. Finally, CNNs have been use on tasks in other fields than computer vision. 1D CNNs have been used for audio and text processing for classification tasks [33].

1.3.5 Datasets

Popular datasets other than the large ImageNet challenge exists and are publicly available for research. Among them a few appear regularly to benchmark deep learning models :

- The *MNIST* dataset [26] is a bank of 28×28 gray-scale images of hand-written digits. It therefore contains 10 classes, 60000 training samples and 10000 test samples. State-of-the-art models typically achieve an error smaller than 0.2 %.
- The *CIFAR-10* [34] dataset consists of 32×32 RGB images with 10 classes mixing vehicles and animals. It contains 50000 training samples and 10000 test samples with balanced classes.
- The *Caltech-101* [35] dataset contains 101 classes of colour images with averages sizes of 300×200 pixels and there are about 40 to 800 images per category.
- The *COCO* dataset [36] is a large-scale image bank for object detection, image segmentation and image captioning tasks with 91 classes and 2.5 million label instances in 328 thousand images.

1.4 Machine learning workflow

Going back to the fundamentals, the training workflow of a neural network essentially consists in a series of simple steps that can lead to the elaboration of a powerful model.

Data preprocessing The dataset available to solve a particular classification task sometimes preliminary transformations in order to be processed by a neural network. Namely, the data sometimes need to be formatted in a tensor format and normalized. Image data may need to be resized to a given input format and their pixel value normalized between 0 and 1. In general, conducting a preliminary statistical analysis of the dataset and its constitutive features can help identify interesting trends as well as problematic samples.

Evaluation method After choosing the appropriate performance metric and loss function, an evaluation method needs to be selected in order to accurately measure the evolution of the training process. The evaluation method dictates how the dataset is partitioned into a training and a validation set. This separation is of the utmost importance in order to avoid a devastating phenomenon in machine learning : *overfitting*. Overfitting manifests itself when a model performs well on the dataset it was optimized on but fails to *generalize* on new input data. Indeed, the essential goal of any model is to be utilized in a real-world application on data never seen before. The model is exclusively trained on the training set and overfitting is therefore detectable by evaluating the model on the validation set. Overfitting starts when the model learns patterns that are specific to the training set and does not generalize to other data, as depicted in Fig. 1.9.

The most effective way to cancel overfitting is to train the model on a larger amount of data. The main patterns will be dominant compared to the specific patterns of the dataset and

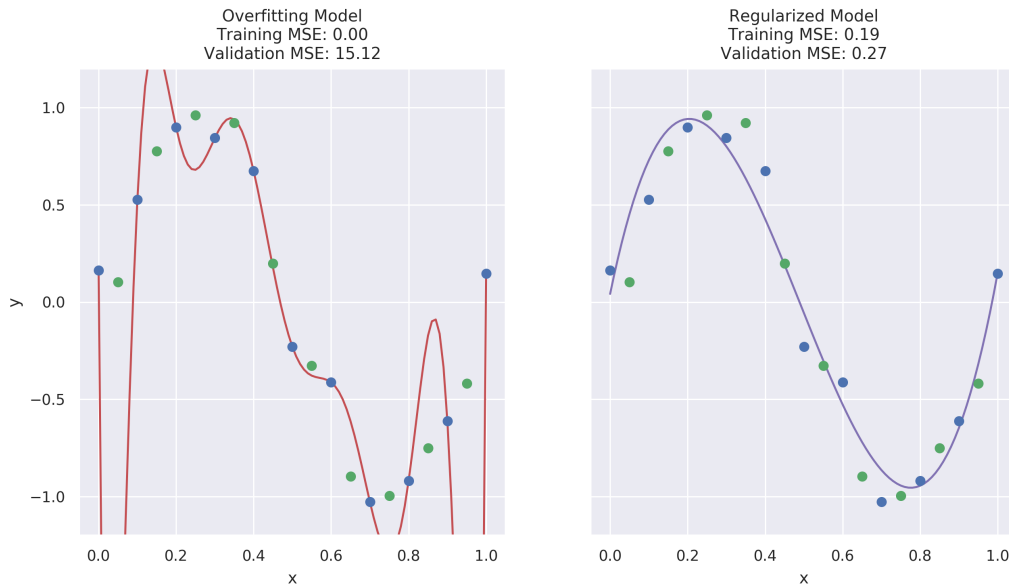


Figure 1.9: Overfitting example with a polynomial model. Blue dots are the training set, green dots are the validation set. The red model to the left interpolates exactly the blue dots with a tenth order polynomial. Hence, it has a training MSE of 0. The purple model to the right is a regularized third order polynomial with much less parameters ; it displays a higher MSE error on the training but generalizes on the validation set with an error in the same order of magnitude.

the model will achieve better generalization. However, this solution is costly and can even be impossible to implement. A second tactic is called *regularization* and involves reducing or constraining the learnable parameters of the model. By limiting the quantity of information a model can store, it will retain better the most significant patterns of the input data, thus reducing overfitting. An overview of most common regularization methods is provided in Section 1.5. Going back to the choice of evaluation method, multiple techniques exist to partition the dataset into a training and validation set.

- A *hold-out* validation set is created by selecting at random a given percentage of the dataset. A 80/20 % repartition between the training and validation set is usually found in the literature.
- A *K-fold cross-validation* divides the dataset into K equally sized subsets. A number K of learning phases is conducted by sequentially validating the model on a different validation set, while training it on the remaining $K - 1$ subsets. This approach allows to obtain a measurement of the training and validation metrics with a low variance but requires multiple trainings of the model, which can be time consuming.

Establishing a model baseline The next step is to create a baseline model that achieves *statistical power*. Indeed the minimum performance metrics achievable over a task are the ones of a random model. Developing a baseline model demonstrates that performances above random guessing are achievable and validates the hypotheses that were made concerning the input dataset.

- The input samples contains information about the targets to predict.

- The dataset contains enough samples to expose the relationships between the inputs and the targets.

Overfitting on the training set This step can be useful to select an adequate range of *hyper-parameters* for the model to be optimized. Hyper-parameters designate all parameters that are not learned via back-propagation but that are still subject to tuning. This includes :

- The number of convolutional and dense layers in the model.
- The number of channels and neurons in the layers.
- The spatial size of the filters.
- The learning rate and optimizer algorithm.

Finding a model that overfits attests that the model has the proper number of parameters to learn patterns in the input data that relates to the target output.

Regularization and hyper-parameter search This last step involves modifying the overfitting model in order to find the best trade-off between generalization and overfitting. This modification involves sweeping hyper-parameters and adjusting regularization methods repeatedly and produce several models. This process can be time consuming, depending on the number of hyper-parameter that are tuned and the duration of a network training. The selection of the best performing model is done based on the validation set performance metrics.

However, because this "hyper-parameter training" is performed on the validation set, a second overfitting phenomenon can occur. Indeed, the selected model can very well fit the data of the validation set but will not generalize on new data. For this reason a third *test set* is necessary to evaluate the performance of the selected model. This test set is a hold-out subset that is selected at the beginning of the workflow and is kept apart until the last moment. This test set is generally given when using a popular dataset in the literature ; It is built to display the same characteristics as the training and validation sets while avoiding any duplicate sample between the two.

1.5 Regularization methods

Network size The simplest way to reduce the overfitting of a model is to limit the number of parameters that can be optimized, otherwise called the model *capacity*. In essence, the more there are parameters, the more it is possible for it to learn useless patterns that are only present in the training set. The ideal network capacity can be searched with an hyper-parameter optimization on the number of layers and neurons in the model.

Early stopping One interesting property of neural networks is that you can limit the amount of patterns that are learned by stopping the training loop when the validation accuracy starts to worsen. This technique is shown in Fig. 1.10.

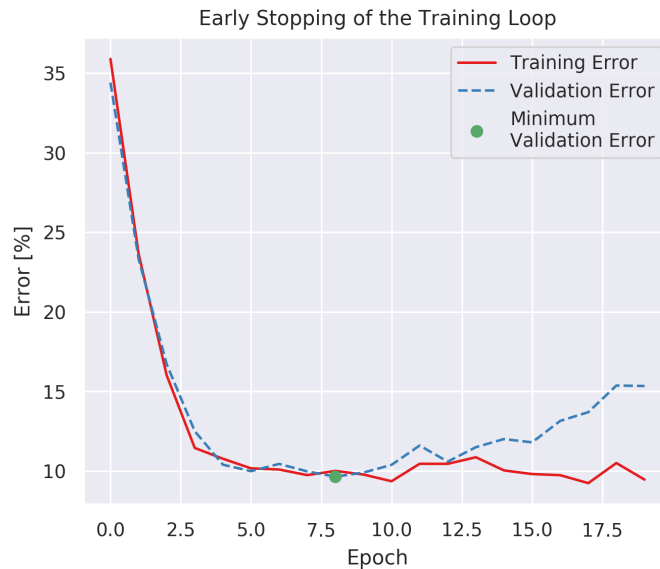


Figure 1.10: Early stopping regularization by monitoring the validation accuracy at each epoch.

Weight regularization Instead of limiting the number of parameters, constraints can be added in order to control the weight distribution. This involves adding an additional term in the loss function that depends on the weight values.

- *L1 regularization*, also called *lasso regression* [37], adds a L1-norm term into the loss function, effectively penalizing large absolute values of weights. In practice, it tends to set the values of some weights to 0 and produces sparse weight matrices.
- *L2 regularization*, also called *ridge regression* [37], adds a L2-norm term into the loss function and penalizes large square values of weights. It effectively shrinks the values of all weights towards 0.

For both solutions, the λ parameter allows to balance the influence of the regularization term in the loss function.

Dropout It is a regularization technique that consists in randomly setting intermediate activation of neurons to 0, effectively "dropping" some neurons in the layer. It is controlled by a *dropout rate* that dictates the proportion of neurons that are dropped. This process is only applied during the learning phase and can be interpreted as an introduction of noise in the model. This effectively breaks complex patterns that are the cause of overfitting [38].

Data Augmentation At last, this technique allows to artificially creates new data samples from the training set in order to promote the generalization of the model [21]. Usually applied

to image data, it modifies the samples by performing a series of random transformations that are controlled by range parameters. It includes :

- Random rotations.
- Random translations in the x and y axes.
- Random scaling.
- Random shearing in the x and y axes.
- Random mirroring on a particular axis.

The transformations often need to fill missing pixels after a rotation or translation with a constant value. With this technique, the network will never process the same samples twice due to the random transformations, it therefore helps to get rid of overfitting.

1.6 Machine Learning Frameworks

1.6.1 Pytorch

With advanced research conducted in the field of deep learning during the last couple of years, dedicated APIs have emerged to offer deep learning training platforms on personal computers. Among them, *Pytorch* [39] is a Python based API that allows an easy prototyping and training of conventional models such as neural networks, CNNs and other advanced topics. It distinguishes itself by providing a very clean interface that can easily integrate customized research models. The Pytorch framework is used in the remaining sections of this work and therefore requires a quick introduction. Pytorch is composed of two main features :

1. At the core of Pytorch is the *Autograd* package. It implements automatic differentiation when performing operations on tensors. The *Tensor* class tracks every operation that is applied to an input tensor in order to build a *computational graph*, similar to what was introduced in Section 1.2.3. The backward pass therefore uses this graph to update the weights of the model. A neat interface is proposed to add customized layers and activations in the framework for research ; it consists in implementing the forward and backward passes of the function, as shown in Fig. 1.11.
2. Pytorch allows to perform complex operations fast using backend libraries, optimized to run on CPUs as well as GPUs. If a CUDA-capable GPU is available, tensor objects can be moved seamlessly to the GPU's memory and the tensor operations are conducted with high performance CUDA C++ kernels of NVIDIA's *cuDNN* library. Moreover, Pytorch allows the programmer to interface its own CUDA kernels in order to perform optimized custom layer operations. A CUDA kernel offers increased performance by leveraging the massive multi-thread capabilities of GPUs : the kernel is called multiple times in different threads and allows to parallelize the operation. Additionally, CUDA GPUs are able to process multiple thread blocks in parallel thanks to several *Streaming Multiprocessors* [40].

```

1  class Exp(Function):
2      @staticmethod
3      def forward(ctx, i):
4          result = i.exp()
5          ctx.save_for_backward(result)
6          return result
7      @staticmethod
8      def backward(ctx, grad_output):
9          result, = ctx.saved_tensors
10         return grad_output * result
11
12 #Use it by calling the apply method:
13 output = Exp.apply(input)

```

Figure 1.11: Customized function integration in the Pytorch Framework, reproduced from [39]. The new function is derived from Autograd’s *Function* class and requires the implementation of the forward and backward methods. The forward method produces the output of the layer from its input and the backward pass computes the gradient of the output with respect to every input. From this information, Autograd is able to generate the computational graph. This example involves reproducing a simple exponential function.

1.6.2 Training wrappers

Additional libraries can be used in the Python framework to improve some specific aspects of the learning. A small list of three libraries that are used in this work is presented.

Pytorch Lightning The Pytorch Lightning [41] allows to better organize Pytorch code into specific categories and therefore provides a clearer view of the training algorithm.

1. The research code regroups the model architecture and configuration.
2. The engineering code regroups the dataset preprocessing, the training loop sequence and the optimizer configuration.
3. The logging code regroups all statements related to the monitoring of the model’s metrics during training. Different *callback* functions allow to upload the training metrics into an online visualization platform such as TensorBoard or WandB [42].

Albumentations The Albumentations library [43] provides advanced image data transformation methods for preprocessing and data augmentation of an image dataset. It provides the performances in terms of benchmarking compared to other image processing libraries.

Optuna The Optuna library [44] provides wrapper functions to train and optimize hyper-parameters of machine learning models. Admissible ranges and sampling functions for hyper-parameters are specified by the programmer and the library generates and trains multiple models on the dataset and reports their performance metrics. The sampling functions selects hyper-parameters from configurable distributions and allow a fine-control over the design space exploration. Optionally, sampling functions can be parametrized on conditional

distributions over previously selected hyper-parameters, in order to implement *meta-heuristic* search algorithms such as *simulated annealing* [45].

The hyper-parameter training process also comprises the use of *pruners* that allow to detect and stop the training of unpromising models in order to save training time.

2.1 Challenges in Embedded Machine Learning

2.1.1 Context

Deep learning advances have unlocked state-of-the-art performances on image recognition tasks but with a significant power cost. The forward pass of a deep neural network is a computationally complex operation, necessitating powerful platforms in order to perform inference tasks on real-world applications. Embedded machine learning is an emerging trend where advanced models are optimized to be fitted inside smartphones, wearables and other Internet of Things (IoT) devices. It enables the possibility of local processing of data for real-time applications and satisfies privacy concerns which arise when uploading raw data into cloud computing servers. Yet integrating large deep neural networks such as VGG-16 in milliwatt platforms is a real challenge and necessitates algorithmic and hardware innovations in order to lower the energy costs. Indeed, typical image and speech processing models require a computing power up to 1 tera-operations per second (TOP/s) [5] and involves managing millions of parameters in memory. While embedded devices achieve performances in the order of hundreds of GOP/W, a large gap separates these devices from running state-of-the-art models. This section presents some optimization techniques that allow to reduce this gap by leveraging the properties of deep neural networks and hardware platforms.

Deep learning models have essentially 3 main characteristics [5] that can be exploited in order to improve their execution time and energy efficiency :

1. The *network structure* of deep learning models can be exploited since convolution and dense layers demonstrate a potential for the parallelization of operations and optimization of data reuse.
2. Deep learning models have shown *robustness against a loss of precision* in its operations. This can be leveraged by specialized hardware with limited computational accuracy.

3. Deep learning networks often manifest weight vectors with a large *sparsity*. It can help reduce memory transfers and the number of operations and can further be stimulated with adequate training techniques.

2.1.2 Network structure optimization

The convolution operation of a CNN layer (Eq. 1.20) is an algorithm requiring a total of 6 nested loop, accounting for the channel, width and height dimensions of the input and output feature maps. A large amount of parameters is therefore only fetched once from memory and reused throughout the innermost loops. The relative ordering of the loops can be modified to promote the data reuse of any of the following parameters :

1. *The input stationary dataflow* indicates a technique where the input spatial and channel indices are located in the outermost loops. The input feature map values are therefore only loaded once from memory. On the other hand, the weight values must be frequently loaded and require a high bandwidth. Additionally, the output feature map is gradually constructed and the intermediate results of the accumulation needs to be written back to memory several times.
2. *The weight stationary dataflow* indicates that the weight values are only loaded once from memory while the input and output values are fetched multiple times.
3. *The output stationary dataflow* puts the output feature map indices in the outermost loops, the output values are therefore accumulated during the iterations of the inner loops and are only written once into memory. Without surprise, this is at the expense of the input and weight values that fetched multiple times during the loops.

The ideal dataflow is chosen based on the relative size of each dimension in order to minimize data fetches. Often, hybrids techniques are used to optimize the bandwidth [46].

The cost of a memory read or write depends heavily on the presence of a memory hierarchy. Cache systems, on-chip RAM and scratchpads close to the processing unit can be leveraged to hold temporary data and minimize slow and energetic accesses to external memory. However, it requires a careful planning of the data movements as all values cannot fit inside those fast access memories. When designed on an FPGA or a dedicated SoC, the size of the memories can be adapted towards the network needs. Conversely, a network can be trained using the available information about the cost of memory accesses in a Multi-objective optimization [47]. Moreover, the convolution operation can also largely benefit from distributed processing across multiple processing elements (PE), each equipped with a small local memory and communicating results with each other. For example, the Eyeriss CNN accelerator [46] develops a new dataflow optimized to work on a systolic array of PEs. In this framework, different CNN configurations can be adapted to use optimally the PE local storage and inter-communication between PEs. Indeed, the flexibility of an accelerated deep learning platform is paramount in order to adapt to various tasks and ever-evolving network topologies. For that reasons, dedicated machine learning hardware often focus on optimizing convolution and matrix multiplication operations in order to keep sufficient design flexibility.

In parallel, advances at the algorithmic level have been achieved, offering minimal complexity for convolutions with small filters using the winograd's minimal filtering algorithm [48].

It allows up to a 7.3x speed-up compared to classical FIR formulations and 2x speed-up compared to FFT techniques.

2.1.3 Exploiting robustness against errors

CNNs have demonstrated a certain tolerance against perturbations on their weights and activation values and can be exploited by purposely limiting the operation's precision in order to save energy and computational time. For the majority of CNN accelerators, floating-point data-types are replaced by fixed-point representations with hardware optimized to perform integer operations. Additionally, it allows a better usage of the memory systems and minimizes the cost of fetching weights and inputs from external memory. For minimal loss of accuracy, the deep learning model can be trained with intermediate quantization of weights at each learning step. At the same time, the weight precision can be tuned based on a modelling of the induced quantization noise in the different layers of the model [49]. As an extreme case, a binary-weight adaptation of AlexNet has demonstrated a loss of accuracy on the ImageNet challenge of less than 1 % [50]. Alternatively, recent techniques have been developed to train a cascade of several models. Simple models would be used in the early stages of the processing chain in order to return a prediction for samples with a distinctive class probabilities, therefore avoiding unnecessary computations with energy-hungry models. Multi-stage techniques have demonstrated to provided superior performances for highly-skewed probabilistic data [51].

2.1.4 Exploiting sparsity

At last, the weights and intermediate activations of deep neural network show interesting sparsity property that can be exploited. Indeed, models such as AlexNet have on average only 70 % its activations that are non-zero [5]. This can be exploited during computations by skipping operations when the input operands are zero [46]. At the same time, compressing the weights in memory allows to optimize the data transfers between memory and processing units [52] but adds compressing and decompressing stages before the layer operations. An interesting alternative is to achieve the compressing directly in the compressed domain using a matrix transform such as the Singular Value Decomposition (SVD) [53]. Regular CNN accelerators can therefore be used with a compressed representation of the network. Again, the sparsity of a model can be promoted during training by regularly pruning the weights [54] in order to save computations in large layers. This technique has been combined with Winograd algorithms to allow a pruning of up to 10 % of the weights without any loss of accuracy on for the LeNet-5 model on the MNIST dataset [55]. Similarly, reductions of the number of multiplication operations in the order of 10x have been achieved by moving the activation function after the Winograd transform with only a loss of 0.1 % on the CIFAR-10 dataset.

2.1.5 Opportunities

Embedding these techniques has allowed to build accelerators achieving performances that exceed 1 TOP/W efficiencies with a throughput of the order of 100 GOP/s [5]. However,

integrating these performances in an end-to-end framework is still a major challenge for embedded deep learning. For computer vision applications, it involves optimizing a complete pipeline comprising efficient image sensors and data transfers to the deep learning accelerator. In parallel, efforts are made to integrate the training phase in the embedded system as well. *Online learning* on an embedded chip allows to continuously optimize the system after launch, based on new data captured by the device.

2.2 Neuromorphic Computing

With a view to further decreasing energy consumption, it is believed that a paradigm shift [7] is necessary for a network computational complexity to be dependent on the input activity and not the network size. Inspiration taken from biological phenomena led to a new approach of information processing inside neural networks, called neuromorphic processing that encodes information in the time domain. This sparse encoding allows to achieve significant reductions in power consumption and latencies compared to conventional deep learning methods and may unlock new learning techniques from biological inspiration [6]. This section provides an introduction to the neuromorphic domain, with an emphasis on computer vision techniques.

2.2.1 Dynamic Vision Sensors

Event-based cameras or *dynamic vision sensors* (DVS) introduce a completely new representation of visual scenes for computer vision applications targeted for real-time and low-power requirements. Instead of collecting luminance information by exposing the pixel array of the sensors during a certain period in order to build a *frame*, pixels of a DVS asynchronously and independently produce an *event* when a brightness change (in log-intensity) crosses a defined threshold. As explained in further details in Section 2.2.3, this process takes inspiration from neurosciences and mimics the process by which biological retinas send visual information to the brain [20]. As shown in Fig. 2.1, this design exploits the fundamental redundancy that exists between frames when capturing video streams with a conventional sensor : only a small portion of pixels undergo a sufficient brightness change between successive frames ; this portion generally corresponds to edges of moving objects present in the image.

DVSs therefore display the following advantages compared to classic image sensors :

1. The output representation only include the addresses of pixels that produced an event. This format called *address event representation* (AER) enables to represent a scene with sparse information spread over time. Data packets contains the (x, y) coordinates, time stamp t and the polarity p of the brightness change in the event's pixel. It leads to a significant bandwidth reduction in the data stream produced by the vision sensor, with an average in the hundreds of kB/s [8]. A DVS is therefore a data driven sensor : the produce event stream depends on the amplitude of brightness changes and movements in the recorded scene. In addition, this reduction in data rate is synonym with a reduction in power consumption ; event-based cameras typically consumes about 10 mW of power.

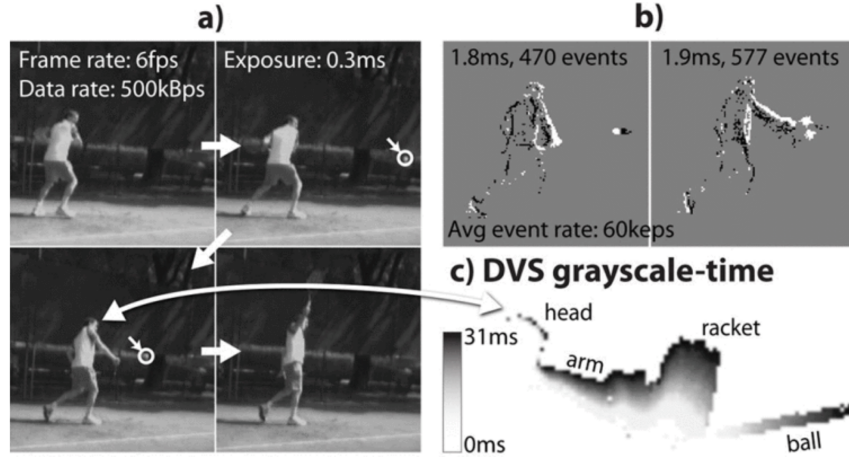


Figure 2.1: Comparison between the output of a standard APS and a DVS, adapted from [9]. The output of a DVS cameras only contains pixel that felt variation in brightness. “(a) Sequence of APS output images with an exposure time of 0.3 ms; (b) Short DVS output time slices of the same scene, white and black pixels show the ON and OFF events; (c) Close-up of the arm, ball and racket over 31 ms time slice using the gray level to show the event time.” – Brandli et al.

2. With its asynchronous mechanism, a DVS sensor displays a time resolution orders of magnitude higher than a conventional frame-based sensor, in the microsecond range [56]. Since it is not necessary to wait for a global exposure time, it provides visual information with temporal precision orders of magnitude higher than a frame-based sensor.
3. DVS sensor architectures possess a logarithmic sensitivity on changes of intensity, allowing to reach dynamic ranges of 120 dB.

These properties therefore render DVS sensors an interesting solution in applications with high-latency constraints and low-power needs such as robotics and real-time tracking of objects for autonomous vehicles. However, it requires a paradigm shift in machine learning techniques to process this new kind of data and take into account the temporal relations that it contains. Different approaches are discussed in Section 2.2.3. The next paragraphs present the popular DVS devices and the current state-of-the-art in terms of resolution, energy consumption and maximal event throughput.

Posch et al. [8] present an asynchronous time-based image sensor (glstis) that implements an AER readout with QVGA resolution (304×240) in a $9.9 \times 8.2 \text{ mm}^2$ chip supplied by Prophesee. Each pixel independently monitors the change of current of the photo-diode and produces ON/OFF events corresponding to an increase or decrease in the photocurrent that reaches a tunable threshold. As the photoreceptor logarithmically responds to intensity, the ATIS sensor has demonstrated to reproduce static scenes with at least a 143 dB dynamic range. The chip is able to reach event latencies below $10 \mu\text{s}$ with an average power ranging from 50 to 175 mW for static and highly dynamic scenes.

An architecture presenting a dual dynamic and active pixel vision sensor (DAVIS) in a 240×180 pixel array is proposed by Brandli et al. [9]. It includes an active pixel sensor (APS) outputting synchronous frames at 50 fps concurrently with an asynchronous AER readout, in a $5 \times 5 \text{ mm}^2$ 180nm CMOS chip. The hybrid form between frame-based and

event-based offered by this sensor allows to use standard state-of-the-art computer vision techniques while providing additional information from the dynamic sensor in-between frames that can be leveraged in tracking applications. The two representations share the same pixel photo-diode and the circuitry for the APS only accounts for a 5 % increase in silicon area. A global-shutter architecture is used for the APS readout in order to suppress artifacts related to moving objects present in a rolling-shutter sensor. The sensor provides a 130 dB dynamic range for the DVS portion and 51 dB for the APS. The asynchronous readout has a 3 μ s latency and triggers for 11 % relative changes of intensity. The power consumption of the DVS portion ranges from 4 to 14 mW depending on the number of events produced. It produces on average 60 thousand events per second (keps) in real-world conditions, corresponding to an event stream of 150 kB/s. For comparison, a frame-based sensor of the same resolution operating at 1000 fps would produce data at a rate of 54 MB/s and would still have a poorer time resolution than the DVS sensor.

B. Son et al. [10] report a DVS sensor with a VGA resolution (640×480) offering a throughput up to 300 Meps in a 90nm 8×5.8 mm² chip. It manages to produce high event rates with an architecture that groups neighboring pixels for the AER readout and processing in parallel ON/OFF events. It achieves 80 dB of dynamic range and a contrast sensitivity of 9 % while consuming between 27 and 50 mW for event rates of 100 keps and 300 Meps respectively.

Most recent advances have been performed by Finateu et al. [11] and include a high definition 1280×720 pixel array achieving up to 1.066 Geps on a 40nm CMOS chip. It achieves a power consumption contained between 32 and 84 mW for a throughput of 100 keps and 300 Meps and it offers a 124 dB dynamic range with 11 % contrast sensitivity.

2.2.2 Event-based datasets

Machine learning tasks in the domain of event-based image recognition necessitate publicly available datasets to benchmark models and cross-examine the results accomplished by the scientific community. The event-based community suffers from the lack of datasets covering versatile domains due to the democratization of event-based sensors that is yet to come. Nevertheless, efforts have been made to generate datasets based on two main techniques :

1. Conversion of existing frame-based datasets into event-based datasets using image processing techniques or event-based cameras. This method allows to compare results of event-based models to traditional computer vision algorithms in the frame-based approach and quantify relative losses of accuracies [57].
2. Capture of real-world scenes with event-based cameras. Unlike the other method, this approach allows to integrate real dynamics present in the recorded phenomenon. As part of the information captured by a DVS is contained in the relative time between events, these datasets are preferred to quantify how an event-based model leverages the time information [17].

Conversion techniques have been used by Orchard et al. [58] to adapt the popular MNIST and Caltech-101 datasets for event-based machine learning using the ATIS camera. In order to recreate accurate dynamics of movements, the camera is mounted on a pan-tilt base and random movements are applied to induce the production of events from the DVS. At the

same time, images of the original dataset are projected on a standard LCD monitor. As a result, the N-MNIST and N-Caltech datasets are produced and contains 300 ms of event data per sample, as shown in Fig. 2.2. The authors of this method argues that it provides better samples than the previously established MNIST-DVS dataset ; it was produced using a fixed camera and a moving sample of MNIST on an LCD screen. However, the LCD screen having a limited refresh rate, saccades were visible in the produced data. On the other hand, N-MNIST also shows predictable patterns as the camera movements are limited in their range and the camera mount has determined axes of rotations.

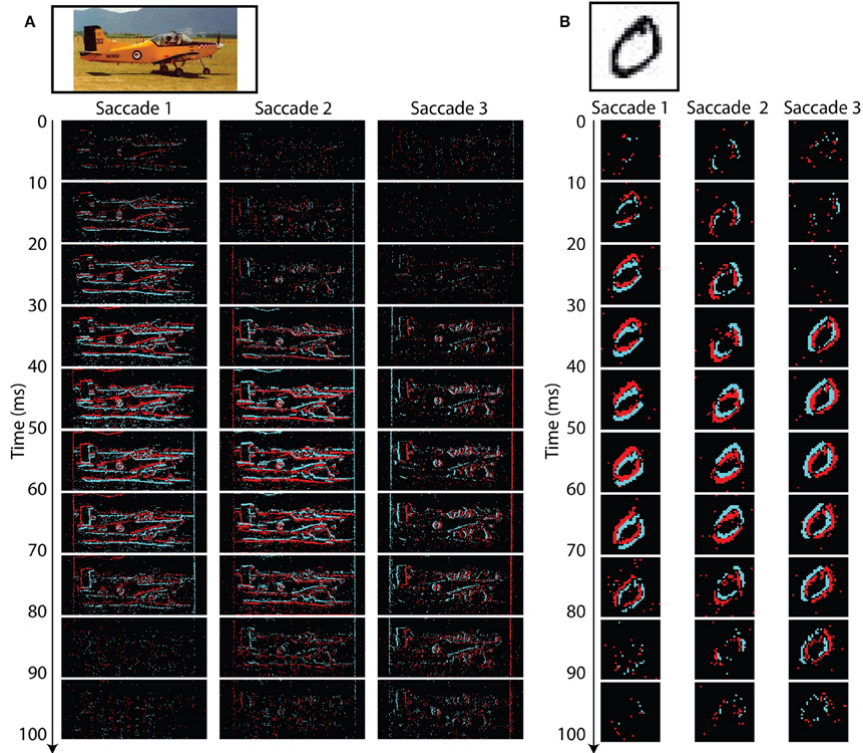


Figure 2.2: Example of produced samples for the N-MNIST and N-Caltech datasets, reproduced from [58]. Samples are created from three saccades applied on a DVS camera set on a pan-tilt mount. Original samples are displayed on top and events associated with three saccades are shown below. Samples are constructed by concatenation of these saccades.

An alternate way to generate event-based samples from conventional datasets is through using processing algorithms that simulates the operation of a DVS. Event rates of pixels are set to be proportional to the gray-scale intensity of the still sample image in the work of Thorpe et al. [59] while O'Connor et al. [60] entirely simulates image motions to generates events. Other popular conversion methods include generating a spike signal from a *poisson* stochastic process where the average firing rate is proportional to gray-scale levels of pixels in the frame-based image. In all cases, a fundamental flaw in this approach is that specificities such as realistic movements and camera non-idealities are not taken into account and models trained on these datasets might not transfer their learning on actual DVS data.

Finally, scenes of the real-world can be recorded with DVSs in order to capture high frequency temporal information contained in the observed phenomenon. Concerned by the lack of such datasets, Sironi et al. recorded the N-CARS that comprises events taken from an urban and highway environment. It is a binary classification problem of 100 ms samples containing cars and background objects. Since classification on the N-CARS dataset is a main subject

of this work, the specificities of this dataset are further examined in Section 3.1. With the same goal, Amir et al. [20] created the DVS-Gesture dataset comprising 1342 instances of a set of hand and arm gestures, recorded with the iniLabs DVS128 camera [61]. There are 11 classes including hand waving and arm rotations, as shown in Fig. 2.3.

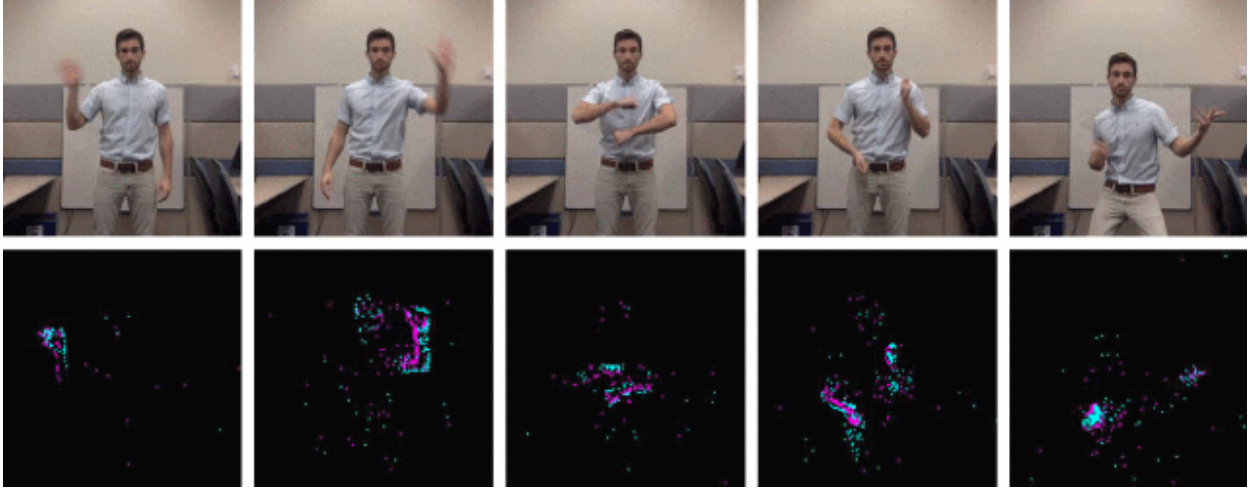


Figure 2.3: Example of samples from the event-based DVS-Gesture dataset along with a frame-based comparison, reproduced from [20]. “Top: 24-bit RGB video frames of the gestures used in this experiment. From left to right the gestures are hand-wave, large-circle, hand-rotate, air-guitar, and air-drums. For compactness, only one hand is shown for hand-wave and large-circle, and only one direction for large-circle. Bottom: DVS ‘frames’ created by superimposing spikes over a 5 ms window aligned to the onset of each RGB video frame.” – Amir et al.

2.2.3 Spiking Neural Networks

As event-based cameras allow to represent visual information in a sparse manner with a high energy efficiency, large efforts have been made to develop processing systems capable of working with sparse information. The brain has provided a source of inspiration in developing machine learning models and has created a versatile field of research named *neuromorphic engineering*, with applications comprising event-based visual and audio sensors [62], event-based processing models inspired from the neuron brain cell and neuromorphic processing chips implementing those models in low-power low-latency platforms. In this section, the core concepts around neuromorphic processing are introduced starting from the definition of the biological neuron. Then several models that takes advantage of the temporal sparse coding are reported along with their training techniques. Finally, several low-power chips are described.

2.2.3.1 Biological neuron behavior

The two main features of a biological brain are *neurons* and *synapses*. A neuron can be interpreted as a processing unit that accumulates information coming from connections it has with other neurons through synapses. Neurons exchange information with each other in sparse electrical signals called action potentials or *spikes*. Neurons accumulate this electrical

potential on their membrane and transmit a spike to subsequent neurons if a voltage threshold is reached. Similarly, the *membrane potential* of *post-synaptic* neurons is raised and the cycles are repeated if a sufficient number of spikes is accumulated over a short period of time [63]. After a neuron has outputted a spike, its ability to produce another one is disabled for a short period of time called the *refractory response* and its membrane potential returns to a *resting voltage*. The behavior of a biological neuron additionally comprises complex stochastic behaviors and complicated response schemes. The information transmitted by a neuron is therefore contained in the time domain since relative spike timings will influence the membrane potential of post-synaptic neurons. In addition, information is also carried by the number of pre-synaptic neurons that act on a post-synaptic neuron. The link with event-based DVS cameras is readily visible as it also relies on a temporal coding of the brightness.

The learning process in the brain is believed to occur thanks to modifications of the synapse behavior and the propagation delays between neurons [64]. At this point, there is already a parallel with the formulation of a fully-connected layer (Fig. 1.1). Indeed, the work of Rosenblatt et al. [23] that formulated the MLP formulation took its inspiration from the neuroscience advances of the time. However, an MLP transmits real value data between layers instead of sparse spikes signals in the time domain. Therefore the main difference that separates the field of neuromorphic computing with classical machine learning is a formulation that tries to better imitate the behavior of an actual neuron in the hope of reaching the performance/energy trade-off achieved in biological systems.

2.2.3.2 Simplified model

With the goal of unlocking increased learning performances compared to conventional neural networks, models have been developed to imitate the biological neuron behavior as closely as possible. However, the various models proposed in the literature always expose a trade-off between biologically plausible behaviors and computational complexity. The selected mechanism to model the neuron is often referred to as the *update* rule of the neuron. The following paragraphs describes one of the most popular model used in neuromorphic computing. The Leaky Integrate-and-Fire model [65] approximates the biological neuron with a first order Infinite Impulse Response (IIR) model described in Eq. 2.1.

$$\begin{aligned}
 u(t) &= \begin{cases} u(t-1) - L + x(t) & \text{if } u(t) \in [u_{\text{rest}}, \theta] \\ u_{\text{rest}} & \text{otherwise} \end{cases} \\
 s_o(t) &= \begin{cases} 1 & \text{if } u(t) > \theta \\ 0 & \text{otherwise} \end{cases} \\
 x(t) &= \sum_{i=0}^{N-1} w_i s_i(t)
 \end{aligned} \tag{2.1}$$

In this formulation, the signal $u(t)$ is the membrane potential of the neuron. At each time-step, the membrane potential is updated with a sum of all (N) synaptic input spikes $s_i(t)$ weighted by a vector of learnable weights w_i . A leakage constant L is included to reduce the membrane potential if no input spike appears during several time-steps and its value therefore relates to the capacity of the neuron to remember the influence of past input spikes.

If the membrane potential crosses a pre-defined threshold θ , its value is reset to a resting constant u_{rest} and an output spike is produced in the post-synaptic signal $s_o(t)$. In addition, the model implements a resting period t_{ref} during which the membrane is maintained to a resting value.

Illustrations for this process can be found in Section 3.2 for another neuron model : the Spike Response Model (SRM) [66] that builds on the LIF by generalizing the computation of the membrane potential to a convolution with a Finite Impulse Response FIR filter. More advanced models include higher order effects such as in the complex Izhikevich model [7] that include the 20 main behaviors observed in biological neurons [67]. They also include models tuned to conserve some mathematical properties such as in the Adapted LIF model [65]. In practice, the implementation of the model in software or on a digital platform requires the continuous time formulation to be sampled with a determined *sampling period* and this aspect is further explained in Section 3.2.1.

In summary, a neural network equipped with this neuron model is called a *Spiking Neural Network* (SNN) and displays the following fundamental differences :

- A new dimension accounting for the time axis needs to be considered. The networks accepts inputs in the form of pike signals, possibly coming from an event-based sensor and carries out computations with explicit dependencies between time-steps due to the update rule. The spike signals are transferred to other layers through connections analogous to synapses with a potential travelling time called the *axonal delay*. It then produces an output spike signal containing processed information.
- Information transmitted in-between layers takes a sparse binary format of *spikes* (or *events*). The neuron possesses an internal state corresponding to the membrane potential and the layer output is produced by comparison with a determined threshold. Additional states and internal mechanisms may be added depending on the complexity of the neuron model. Among others, the LIF or SRM neuron can therefore be interpreted as a new choice of *activation function* in the classical formulation of a neural network.
- Unlike neural networks equipped with a sigmoid or ReLU activation, the thresholding operation realized to produce spikes is non-linear and its gradient is not defined. Popular training techniques such as back-propagation therefore cannot be used as they are and alternative solutions must be found. New solutions can be investigated in the context of neuromorphic engineering to develop algorithms that are biologically plausible and may offer better learning performances.

Due to the event-based nature of the produced output spike signals, encoding of information can take several forms :

- In rate-based coding, the information transmitted inside the spike signal is contained in the number of produced spikes, also called *firing rate* measured in spike/s or events/s (eps). This is essentially the coding method that is used in a DVS camera [8].
- In the situation where several spike signals are returned from multiple output neurons, population coding stores the relevant information in the number of neurons that fire at the same time. [68]

- Temporal coding stores the information in the relative position of spikes in the spike signal. It generally achieves very high levels of sparsity as the information can be stored in the time latency of the first output spike. [69]

Due to this architecture, a main property of an SNN is that the number of computations can be voluntarily restrained in the time domain to already output a prediction when the input sample has not yet finished to be processed. A trade-off between the loss of accuracy and the energy spent on computations can therefore be achieved. This process is demonstrated on the DVS-Gesture dataset in Fig. 2.4 with the Slayer SNN model [13].

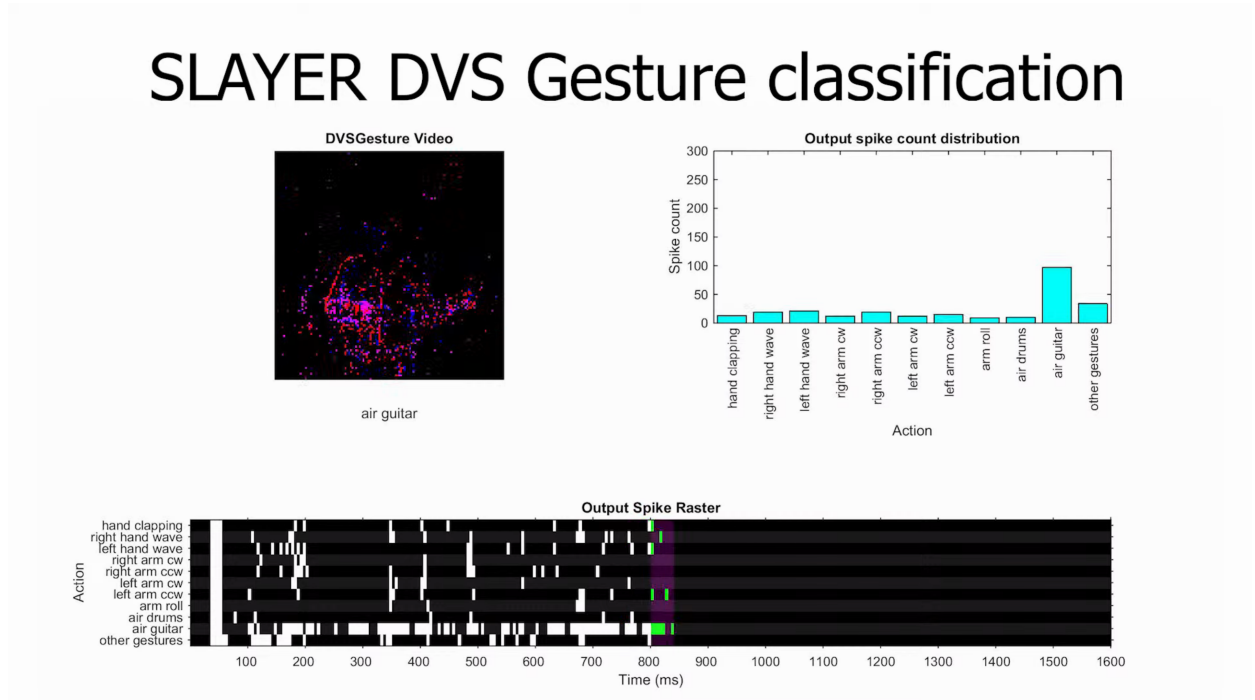


Figure 2.4: Slayer [13] output prediction on a DVS-Gesture Sample. Top left : simultaneous events at a given time-step. Top right : spike count distribution of the output layer (prediction). Bottom : outputs spikes of each neuron of the last layer vs. time. Even if only half of the input events has been processed, the network is already able to output the correct prediction with significant discrimination, based on the accumulated number of spikes.

2.2.4 Learning techniques

In the field of neuromorphic computing, four main trends are found in state-of-the-art techniques to train an event-based model.

Unsupervised methods Unsupervised methods happen in the spiking domain and are biologically inspired. They generally promote correlated activity between pre and post-synaptic neurons to stimulate cooperation. These training methods are often referred to as *learning rules* and notably include Spike-timing-dependent plasticity (STDP) [15] and Spike-driven synaptic plasticity (SDSP) [16].

Handcrafted features Handcrafted feature design comprises an set of techniques to convert the input spike signals into a compressed dense representation by leveraging its spatio-temporal attributes. For example, handcrafted features of traditional computer vision such as corner detectors, line detection and optical flow techniques [17] have been adapted to the event-based domain. Alternatively, new representations designed in a bottom-up approach introduces the concepts of *time surfaces* [70]. Hierarchies of event-based Time-Surfaces (HOTS) combine unsupervised clustering of neighboring events in the spike signal stream and smoothing with exponential kernels to produce dense features that are fed to a Support Vector Machine (SVM), a conventional ML classifier.

In a similar approach the Histogram of Averaged Time Surfaces (HATS) method [17] adds memory units to better leverage the temporal information and regularizes the method to obtain denser and faster representations. The computation of time surfaces can be interpreted as a local convolution of the spike signal with an exponential kernel [18], which shares similarities with the expression of a membrane potential. This technique has demonstrated accuracies of 99.1 % and 90.2% on the N-MNIST and N-CARS datasets, offering competitive performances with deep SNNs.

Other techniques involve the use of predefined (untrainable) Gabor filters [71] [17] in convolutional layers for feature extraction while the training is limited to a regular classifier (linear SVM) to output predictions.

ANN to SNN mapping Mapping of an SNN to a dual ANN model alleviates the difficulty of back-propagation in SNNs by exploiting intrinsic relations that exist between a given SNN and an identical ANN architecture. Indeed, there exists a tight correspondence between the ReLU activation function and rate-based LIF models since activations in the ANN can corresponds to a given firing rate in the SNN model. This approach therefore involves the training of a standard ANN or CNN model on a dataset that is converted from the event-based form to a frame-based form. In vision application, it is typically realized by accumulating the events of the DVS camera to recover the absolute values of pixels. On the one hand, this method loses sensible information contained in the relative timing of spikes. On the other hand, it can leverage knowledge acquired in the deep learning field.

The critical part of this approach is to constrain the optimal weights of the ANN architecture in order to meet a correspondence between the activations of the ANN and the firing rates of the SNN. The activations must be contained between 0 and a maximal value $r_{\max}$ determined by the sampling period of the SNN. This step of the process is often referred as *weight/threshold* balancing as the relative proportion between the weights and the neuron threshold sets the magnitude of the firing rate.

Rueckauer et al. [12] theorizes the link that exists between ANN activations and firing rates and develops a weight normalization method to maintain layer activations in the interval $[0, r_{\max}]$. Instead of normalizing the weights with respect to the maximal value reached in a layer, their technique analyses the distribution of activations to select a normalization with respect to a high percentile. It therefore avoids extremely low average firing rates induced by outliers in the layer activations while minimizing the distortion for high firing rates. Using a CNN model on the CIFAR-10 dataset (converted to an event-based representation using a Poisson process [12]), a weight normalization the 99.9 percentile showed an accuracy drop of only 0.04 % compared to the equivalent ANN.

Alternatively, threshold balancing is achieved by Diehl et al. [72] in order to convert a fully-connected neural network of a sentiment analysis task into a SNN fitted to run on the TrueNorth [73] low-power platform. In his Master Thesis, T. Pirson proposes to find an ideal set of thresholds with a grid search algorithm that minimizes a difference metric between feature maps of the ANN and reconstructed frame-based feature maps of the SNN [65]. It achieves a 5 % loss of accuracy for the conversion of a CNN model on the N-CARS dataset.

Back-propagation in the spiking domain Finally, back-propagation in the spiking domain can be achieved provided that a solution around the derivation of the non-linear spike function is found. From that point, the training algorithm can essentially follow the steps established in Section 1.2.3 to build the learning rule.

The work of Zhang et al. [74] establishes a spike-train level back-propagation (ST-RSBP) applicable to LIF-based deep recurrent SNNs. It realizes the chain rule thanks to a polynomial model of the derivative fitted on simulations of dependencies between pre-synaptic and post-synaptic signals. It achieves a 99.6 % accuracy on the MNIST dataset, converted with a Poisson process.

Other methods such as SpikeProp [75] introduces a derivative term that is a function of the firing time of the spike signals in order to avoid the non-linearity.

Lee et al. [14] treats the membrane potential as a differentiable signal thanks to a stochastic approach where discontinuities at spike time are considered as noise.

Amir et al. [20] proposed an implementation that solves the DVS-Gesture task using the TrueNorth specialized low-power platform [73] ; it involves a back-propagation algorithm on a spiking CNN based on an approximation of the step function’s derivative between the membrane and the spike signals. A triangular function is used instead of a Dirac Delta in order to alleviate the discontinuity. The output classifier is realized using a Winner-Take-All (WTA) algorithm that selects the output class based on population coding.

Similarly, the Slayer [13] model of Orchard et al. proposes a back-propagation algorithm based on a substitute derivative function in order to overcome the spike function non-linearity. It further takes into account the time relations between the membrane potentials and the spike signals across multiple time-steps in the learning rule. CNN models including the Slayer neuron reach state-of-the-art accuracies on the MNIST, N-MNIST and DVS-Gesture datasets, as shown in Table 2.1.

A complete description on the mechanisms of Slayer for the forward and backward passes is given in Section 3.2. For the rest of this work, this algorithm is used for experimentation. It was chosen based on its reported performances, its simple formulation and its publicly available Pytorch implementation. This work further focuses on bringing a real-time solution to the N-CARS classification task ; unfortunately, the authors of Slayer did not reported any testing for this dataset. Nevertheless, several other models have reported the performances of Table 2.2.

Dataset	Method	Accuracy
MNIST	Lee et al. [14]	99.31 %
	Rueckauer et al. [12]	99.44 %
	Slayer CNN	99.36 $\pm$ 0.05 %
	ST-RSBP [74]	99.57 $\pm$ 0.04%
N-MNIST	Lee et al. [14]	98.36 %
	HOTS [70]	80.8 %
	HATS [17]	99.1 %
	Gabor-SNN [17]	83.7 %
	Slayer CNN	99.2 $\pm$ 0.02%
DVS-Gesture	Amir et al. [20]	91.77 %
	Slayer CNN	93.64 $\pm$ 0.5 %

Table 2.1: Slayer performance comparison over the MNIST, N-MNIST and DVS-Gesture datasets.

2.2.5 N-CARS state-of-the-art

Focusing now on the N-CARS dataset, Rebecq et al. took a radically different approach on the N-CARS task and proposed to reconstruct a frame-based dataset in order to apply off-the-shelf powerful deep learning classifiers to solve the task. They use ResNet-18 model to achieve 91.0 % of accuracy.

Similarly, Gehrig et al. proposed a general framework to convert event streams into dense tensor representations called Event Spike Tensors (EST) obtained in two steps. First, a temporal convolution with a fixed kernel smoothes the input spikes. Then, a 3D convolution between the obtained tensor and learned kernels produces the EST. The tensor is then fed to a deep learning ResNet architecture [4] to produce the classification result. Their method further evaluated the computation time necessary to process the 100 ms samples of N-CARS. The EST construction is performed on an Intel i7 CPU (64bits, 2.7GHz and 16 GB of RAM) while the inference is computed on a powerful GPU (GeForce RTX 2080 Ti) ; it yields an average latency of 6.87 ms.

In his Master Thesis, T. Pirson [65] reports unpublished results from Prophesee [76] achieving 95.5 % accuracy with an SNN trained with back-propagation, composed of 2 convolutional layers and 2 dense layers. Unfortunately their experimental setup and training method are not shared.

Recent work on FPGA designs include an optimized implementation of the HATS [17] algorithm by Sethi et al. [19]. In general, the potential of FPGAs resides in the versatile aspect of the platform. Low-power FPGAs can implement an optimized event-based architecture at the hardware level and achieve intermediate performances between general-purpose CPUs and neuromorphic chips. Moreover, FPGAs generally benefit from a configurable connectivity that allows fast integration into an end-to-end embedded system. Their work reports a 3.3 ms latency between a received input event and the classification result with an average power consumption of 620 mW. The formulation of HATS was enhanced by switching computations between histogram normalization and SVM operations to allow factorization in partial sums and alleviate memory requirements. They further provide a continuous formulation where input events are streamed through the processing chain and the result of the computation is

added to contribute to the output classification result. The computation precision is also reduced to 24-bit fixed-point numbers with only limited rounding errors. Unfortunately, the final accuracy achieved on the N-CARS dataset is not reported.

Dataset	Method	Accuracy
N-CARS	HOTS [70]	62.4 %
	Gabor-SNN [17]	78.9 %
	Grid-search AIF [65]	81.97 %
	HATS [17]	90.2 %
	Rebecq et al. [77]	91.0%
	EST [18]	92.5 %
	Prophesee	95.5 %

Table 2.2: Reported accuracies for the N-CARS dataset.

2.2.6 Neuromorphic chips

The last section of this overview on neuromorphic processing involves the analysis of specialized hardware that implements spiking neural networks and other variants. Indeed, the main asset of event-based computing is its potential for sparse layer operations and distributed processing of the update rule for each neuron ; it could be leveraged in a dedicated platform, possibly directly connected to an event-based sensor. Key requirements include a compact design, low-power per spiking operation, programmability of different network architectures and online learning in order to adapt in real-world environments. Neuromorphic processors may come in two types depending on their neuron model implementation : analog and digital implementations. Analog mixed signal implementations include the Neurogrid [78], BrainScale [79], ROLLS [80] and DYNAPs [81] chips while purely digital chips are TrueNorth [73], Loihi [82], ODIN [7] and MorphIC [83]. In addition software platforms like SpiNNaker [84] propose simulation of neurons on general-purpose ARM processors in order to maximize flexibility at the cost of hardware performance. Research on this topic is at its very early stages and processors are mainly supplied to research partners to conduct experiments.

The Loihi chip in 14nm FinFET [82] designed and manufactured by Intel features up to 1 million synapses per core and 128 cores per chip in a Network on Chip (NoC) architecture. This architecture extends in 4 directions to allow other chips to connect in a mesh structure. It comes in packages of 2, 4, 32, 64 and 768 chips with a DVS camera interface for embedded applications. Its online learning rule is flexible for custom designs but notably includes STDP and reinforcement learning reward methods. It implements the alternative CUBA LIF neuron model which involves a synaptic membrane and a synaptic current system of equations ; it further includes configurable synaptic, axon and refractory delays. The energy per synaptic operation (SOP) is reported to be 23.6 pJ.

The TrueNorth chip from IBM [73] in 28nm CMOS emulates 1 million spiking neurons and 256 million synapses distributed across 4096 cores ; it features a cross-bar architecture in a 256×256 format for I/O connections between neurons. However, it requires offline training on a GPU since it does not implement online learning. It consumes about 100 mW of power and is ideal for low-power embedded applications. To that effect, it comes in different packages from the manufacturer, with either 1 or 16 TrueNorth chips. It is interfaced with a dual-core

ARM Cortex-A9 processor and programmable FPGA fabric for maximal flexibility. The reported energy per SOP is 26 pJ. As mentioned in the last section, an end-to-end embedded classification system has been implemented by Amir et al. [20] on the DVS-Gesture dataset. It reports power consumptions of 178.8 mW and an average latency of 104.6 ms between the first event fed to the chip and the output event of the model that sets the final classification result. Similarly the average latency until the last output event of the sample is 120.6 ms.

At UCLouvain, Frenkel et al. developed several neuromorphic chips with different features. An Online-learning Digital spiking Neuromorphic (ODIN) processor was realized in 28nm FDSOI CMOS technology [7] and includes 256 neurons that can be parametrized to implement the Izhikevich behaviors. It has an AER interface and contains memory for 64 thousand synapses and have an energy per SOP equal to 9.8 pJ at 0.55 mV of supply voltage. Online learning is assured with the SDSP rule [16].

MorphIC [83] further implemented a 4 core architecture in 65nm CMOS with a total 2048 LIF neurons and 2 million synapses with a stochastic version of the SDSP learning rule. It stores binary weights in on-chip memory for energy efficiency of accesses and quantization-aware training is used in order to minimize the reduction of network accuracy. It spends 30 pJ of energy per SOP and around 2.3 mW of idle power at 55 MHz.

2.2.7 Summary

Event-based processing is a subject that spans many fields, from image and audio recognition to embedded systems and real-time processing. Many designs explore different trade-offs with a focus on low-power, low-latency or better learning performances with brain-inspired learning techniques. Event-based computer vision is in its early stages and can sometimes suffer from an unfair comparison with well-established frame-based techniques [6]. Nevertheless, DVS sensors show a promising potential for low-power, low-latency and high-dynamic range compared to conventional cameras. The development of biologically-inspired systems is a challenging research field and many solutions are proposed to compete with main stream deep learning methods. SNNs and other event-based models stand out due to their capacity to directly transform data streams coming from event-based sensors in an end-to-end embedded system. Redundant information is filtered at the sensor level and information is directly transmitted with low bandwidth to dedicated asynchronous neuromorphic chips with the AER protocol. Many challenges remain ahead to achieve biological efficiency and performance.

CHAPTER 3

TRAINING & ANALYSIS OF CONVOLUTIONAL SPIKING NEURAL NETWORKS ON THE N-CARS CLASSIFICATION TASK

After an introduction on machine learning and neuromorphic systems, this chapter focuses on training an SNN to solve the N-CARS classification task. In the last chapter, the Slayer [13] model proved to reach state-of-the-art performances on several neuromorphic datasets with a flexible neuron model and unique back-propagation formulation in the spiking domain. The capabilities of this model are therefore carefully analysed in this chapter. First, an extensive analysis of the N-CARS dataset is conducted in order to capture information that will help train and optimize the Slayer model. Then, the constitutive equations and back-propagation algorithm of the Slayer model are introduced with all related hyper-parameters. Finally, a CNN is optimized starting from the Pytorch implementation of Slayer. Elements of the original implementation are introduced and additional features are integrated to create a complete deep learning framework. Automated experiments are conducted to tune hyper-parameters in order to find an optimized model for N-CARS. Properties of the optimized model are then derived to measure the achieved levels of sparsity.

3.1 Analysis of the N-CARS Dataset

This section provides a detailed analysis of the N-CARS dataset established by [17] with the goal of discovering interesting properties that could help to train an SNN for a classification task.

3.1.1 Context

The N-CARS dataset is a two-class dataset composed of several 100 ms samples captured with ATIS camera [8] and stored in the AER format. The samples therefore contain multiple events that comprise a time stamp, an x and y pixel position and an ON/OFF polarity.

The samples correspond to the *Car* class if a car is present in the sample, otherwise they correspond to the *Background* class.

The dataset is built from approximately 80 minutes of video recording of urban and highway environments filmed from the windshield of a vehicle. The samples are constructed with the methodology described in Fig. 3.1.

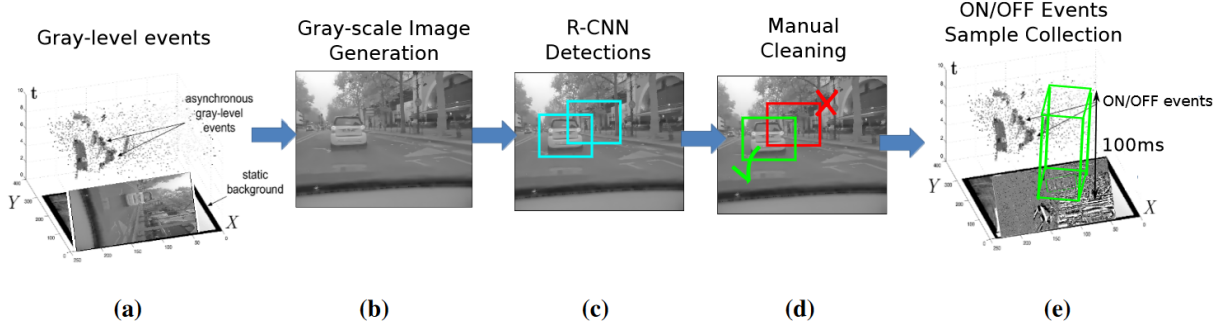


Figure 3.1: N-CARS labelling protocol, reproduced from [17]. “(a) Asynchronous gray-scale events are generated by the ATIS camera. (b) By cumulating gray-scale events, we create gray-scale images every 100ms. (c) We run state-of-the-art R-CNNs to automatically detect bounding-boxes contains cars or background samples. (d) The bounding-boxes are manually cleaned to ensure the dataset contains only correctly labeled samples. (e) The spatial bounding-box is converted to a spatio-temporal bounding-box of 100 ms. Each event stream contained in the spatio-temporal bounding-box corresponds to a sample in the N-CARS dataset.” – Sironi et al.

Gray-scale images are constructed from the AER by cumulating a measurement of the absolute luminous intensity provided by the ATIS camera, over a period of 100 ms. Starting from those frame-based images, an object-detector algorithm is used to extract regions of interest containing either a car or a part of the background. The model used is a state-of-the-art R-CNN model [32], [31]. The extracted regions, or *bounding-boxes*, are classified by the model and are assigned a confidence score on the prediction. Bounding-boxes with an insufficient score are discarded as well as boxes that do not respect certain format constraints. A manual process then comes into play and each sample is reviewed to verify that the predicted label is correct and that the bounding-boxes correctly contain the object of interest. At last, the events associated with the corresponding spatial region of the bounding-box are stored. The time stamps of the sample are shifted in order for the first event to have a time stamp equal to zero. An example of recorded sample of the *Car* class is shown in Fig. 3.2.

The AER format stores each event in a 32-bit word with the following repartition :

- The time stamp is stored in 16 bits.
- The polarity is stored using 1 bit.
- The x and y pixel positions are stored using 14 bits each.

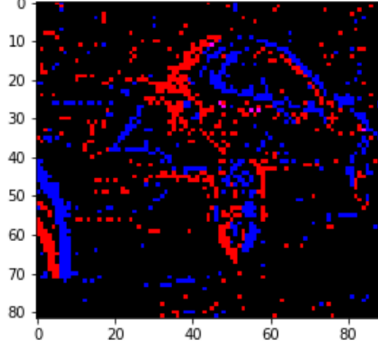


Figure 3.2: The image show events accumulated over a period of 4 ms, red pixels indicate ON events, blue pixels indicate OFF events and pink pixels indicate a simultaneous ON/OFF event.

3.1.2 Dataset Statistics

This section aims at analyzing the N-CARS dataset from a statistical perspective. As detailed in Table 3.1, the dataset is divided into two classes and two subsets for training and testing of a machine learning model. The dataset comprises in total 24029 samples, the *car* class is dominant and contains 51.33 % of the samples. This is therefore the baseline accuracy that any model wishes to outperform for classification of the dataset. The training set contains 15422 samples and the test set contains 8607 samples.

Statistic	Car class	Back. class	Training Set	Test Set
Count	12336	11693	15422	8607
Percentage	51.33	48.67	64.18	35.82

Table 3.1: N-CARS dataset division.

3.1.2.1 Sample distributions

As the samples are derived from the output of an R-CNN model, the image size corresponds to the bounding-box of an object detector. The image sizes are different for every sample and their distribution can be computed for the training and test sets. The conditional distribution of the width, height and number of events of the samples for the two classes are shown in Fig. 3.3. The corresponding statistics are computed in Table 3.2. Note that the x-axis is different for every graph and the quantity of interest is specified in the figure title ; the y-axis is not represented as the quantities shown are distributions.

Because of limited computational resources, the distribution graphs are computed for a subset of the training and testing set comprising 1 thousand samples each, with an equal probability for each class. This subset is referred as the *1K* subset. The statistics computed in the tables however, are done using the full dataset. In this way, maximum, minimum and average values are correctly reported while the main tendencies are displayed in the distribution graphs. Furthermore, this subset is used again in Section 3.4 for the training of an SNN model.



Figure 3.3: Distribution for the width, height and number of events for each class in the $1K$ subset of N-CARS.

Statistic	Class	Width	Height	Events
Mean	Back.	65.85	60.27	3717.04
	Car	65.67	51.63	4202.28
Std.	Back.	27.19	16.55	4291.17
	Car	18.38	15.72	4281.47
Min.	Back.	18	22	500
	Car	24	21	500
25 %	Back.	38	48	994
	Car	52	40	1362.75
50 %	Back.	62	58	2056
	Car	62	48	2812
75 %	Back.	91	70	4807
	Car	78	62	5383.25
Max.	Back.	120	100	59249
	Car	120	100	45885

Table 3.2: N-CARS sample statistics per class on the full dataset.

This shows that several biases are present in the dataset :

- The average number of events for the *Car* class is 13.05 % higher than the *Background* class. This could be useful to know when developing the real-time SNN algorithm in Chapter 5.
- The number of events per sample has a highly skewed distribution, the maximum number of event is in the order of 50000 while the third quantile is only at 5000. The presence of such *burst* samples could be due to a large change of luminosity during the capture of the sample. The test set contains a larger portion of such cases.
- The *Background* class has on average a larger height.
- The shape of the distributions for the height and width of samples is noticeably different between the training and testing sets.

The distributions for both classes together are shown in Fig. 3.4, with the corresponding statistics computed in Table 3.3. It calls also for several remarks :

- The width is contained between 18 and 120 pixels with an average of 66 pixels.
- The height is contained between 22 and 100 pixels, with an average of 56 pixels.
- The differences in distribution shape is readily visible for every quantity between the training and test sets. This could ultimately intervene in the variation of the a model's accuracy on both sets.

Statistic	Width	Height	Events
Mean	65.76	55.83	3966
Std.	23.09	16.70	4292.96
Min.	18	22	500
25 %	48	43	1168
50 %	62	53	2455
75 %	84	67	5108
Max.	120	100	59249

Table 3.3: N-CARS sample statistics on the full dataset.

In order to better understand the origin of the burst samples, the plot in Fig. 3.5 compares the number of events with the image size in pixels, for every sample. It is shown that the burst samples are not related with a larger image size but rather appear for samples with averaged shapes.

3.1.2.2 Event distributions

Distributions can also be computed at the event level, in order to infer statistics about the time stamp, polarity and location of each event. The distributions in Fig. 3.6 are computed again using the samples of the *1K* subset. In order to compute the distributions,



Figure 3.4: Distributions of the width, height and number of events in the $1K$ subset of N-CARS.

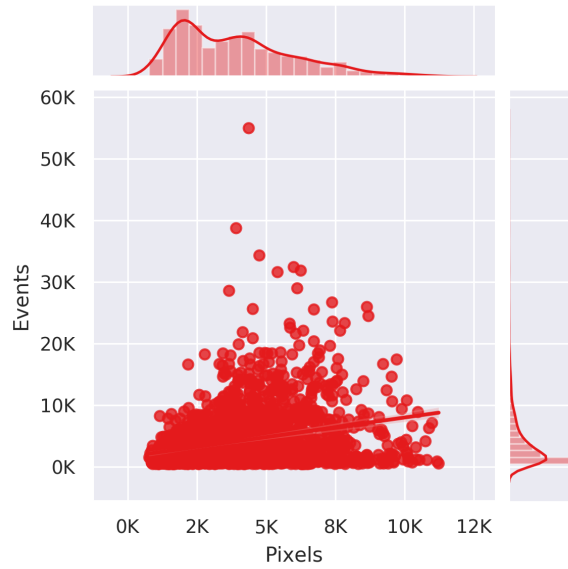


Figure 3.5: Correlation between the number of events and the sample's image size (pixels) in the $1K$ subset of N-CARS.

quantities are weighted by the number of events in the corresponding sample. Subsequently, the obtained distributions relates the average distribution obtained independently for each sample. The *Width Loc.* and *Height Loc.* quantities are computed using the relation in Eq. 3.1.

$$w_{\text{loc}} = \frac{x_{\text{event}}}{w_{\text{sample}}} \quad (3.1)$$

$$h_{\text{loc}} = \frac{y_{\text{event}}}{h_{\text{sample}}}$$

These quantities therefore inform us on the relative position of the event inside the sample image.



Figure 3.6: Distributions of the time stamps, polarity and location of the events in the *1K* subset of N-CARS.

It demonstrates the following facts :

- The time stamps are spread uniformly across the whole sample duration (100 ms). A priori, there is therefore no interest in truncating the temporal dimensions of the samples into a particular time window.
- Polarities have overall equal probabilities. The training set polarity for the *Car* class is slightly biased towards the ON polarity.
- The events are spread rather uniformly across the width dimension. They are slightly more concentrated towards the center for the height dimension. Again, this indicates that no truncation in the spatial dimensions should be carried out a priori.

Considering each individual pixel in a given sample, the distribution of Fig. 3.7 displays how many events are outputted on an average. In other words, it shows the *firing rate* for each pixel.

It reveals that this distribution is highly skewed towards zero, with a proportion of 54.23 % pixels that only fire once and 75 % that fire at most two times. As previously shown, some



Figure 3.7: Distributions of the number of events in a sample for a single pixel in the $1K$ subset of N-CARS.

burst samples introduces a maximum firing of 152 events that is quite distant from the third quartile.

For the events that fire multiple times, the distribution of the time difference between successive events is shown in Fig. 3.8. This distribution is also highly skewed and indicates that only 25 % of the pixels that fire more than once are separated by less than 3 ms. The statistics of both Fig. 3.7 and 3.8 are shown in Table 3.4.

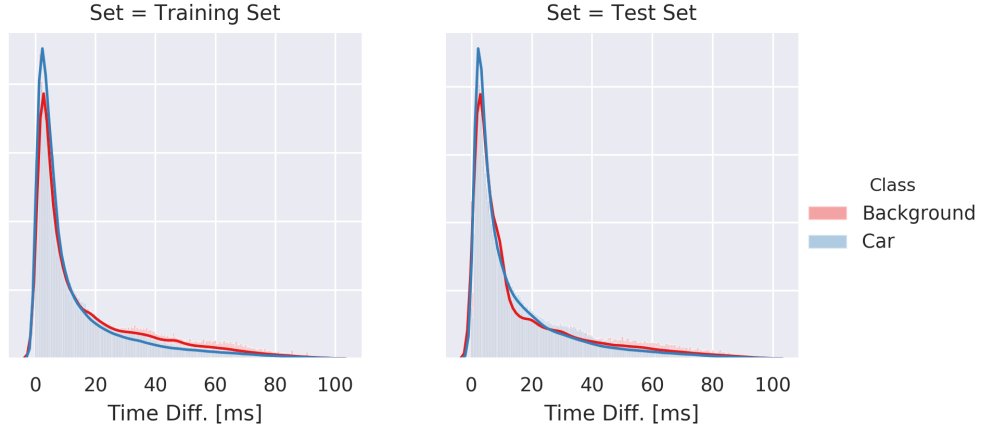


Figure 3.8: Distributions of the time between events for pixels that output more than one event, computed for samples in the $1K$ subset of N-CARS.

This information is useful to understand the side effects that can appear when using a spiking dataset in an SNN model. Architectures like the ones discussed in the next sections perform a rounding of the time stamp in order to represent the events in a multi-dimensional tensor. We say that the events are *sampled* in the time dimension with a given *sampling period*. If multiple events follow each other on the same pixel, they are merged together when truncating the time stamp. This could result in a loss of information if the firing rate of the samples is high compared to the sampling period of the model.

Fortunately, the statistics of this work show that only a very limited proportion of events are merged. As a matter of fact :

Statistic	Event count per pixel	Time difference [ms]
Mean	2.05	15.09
Std.	2.05	17.94
Min.	1	0.01
25 %	1	3.07
50 %	1	7.39
75 %	2	20.36
Max.	152	99.94

Table 3.4: Statistics of the number of events and the time difference between events for a single pixel for samples of the N-CARS dataset.

1. Only 45.77 % of neurons fire more than once.
2. Only 3.31 % of them are merged together when considering a sampling period of 1 ms.

What these statistics do not show is that events that are not generated by the same pixels are going to lie in the same *time-step* for a given sampling period. The distribution to the left of Fig. 3.9 shows the number of events that share the same time-step if a 1 ms sampling period is applied. Equivalently, the distribution on the right divides this number of events by the total number of pixels of the corresponding sample. This results in a percentage representing the sparsity distribution of a given time-step. The corresponding statistics are computed in Table 3.5. To avoid any confusion, the sparsity measure represents "how empty" the images are compared to their size. A low sparsity of a few percents therefore represent an input with very few events compared to the number of pixels in the image. This definition is used again throughout this work.

Statistic	Events per 1 ms	Sparsity [%]
Mean	41.10	0.56
Std.	51.03	0.59
Min.	1	0.004
25 %	11	0.18
50 %	24	0.40
75 %	53	0.77
Max.	1974	25.83

Table 3.5: N-CARS event statistics on the *1K* subset.

For a 1 ms sampling time, the average number of events per time-step is equal to 41. This corresponds to a 0.56 % average input sparsity. The sparsity distribution is skewed and the major part of samples have a sparsity below 1.5 %. This result can be displayed differently in Fig. 3.10, where the average sparsity for every time-step is drawn for the entire duration of a sample. It synthesizes the tendencies previously found in the other distributions :

- The events are uniformly spread across the entire duration of the sample.
- The *Car* class has a higher spiking rate than the *Background* class in the training set. This behavior is not represented in the test set.
- On an average, the spiking rate is around 0.56 % for 1 ms time-steps.

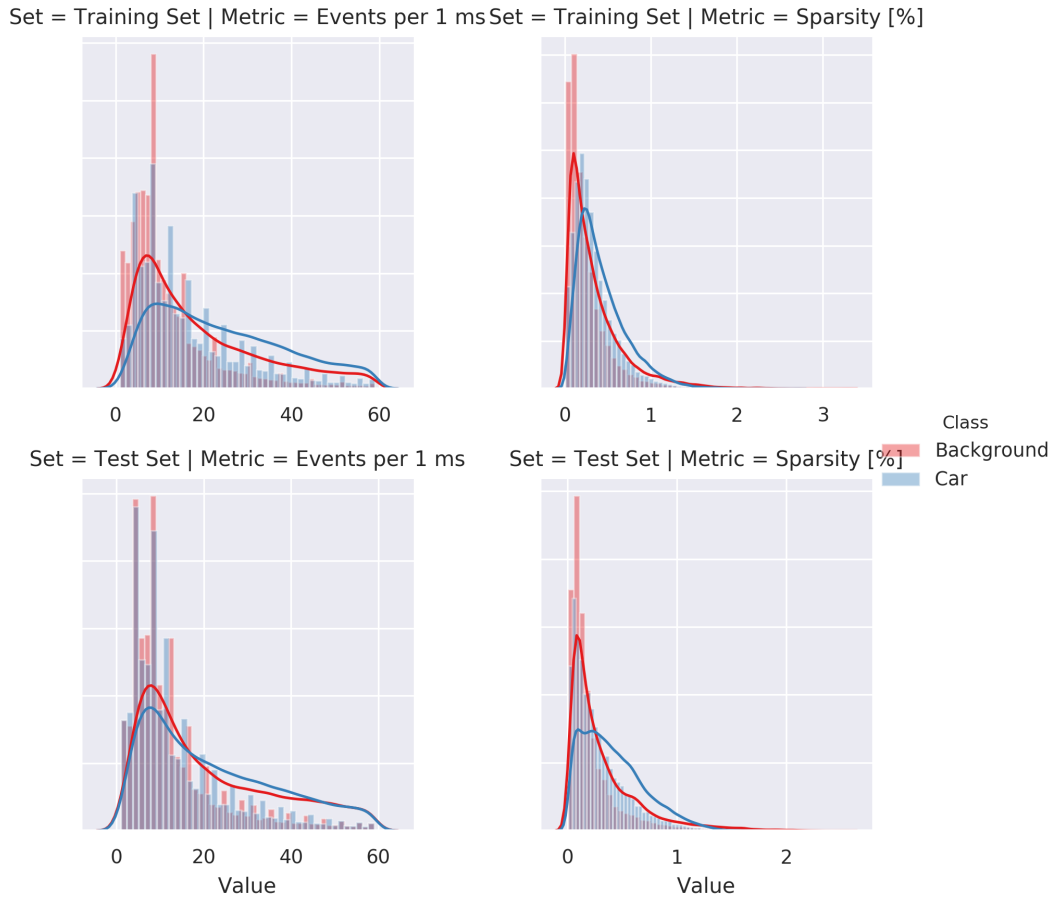


Figure 3.9: Distributions of the number of events in 1 ms time-bins and the corresponding percentage with respect to the sample image size, for samples in the *1K* subset of N-CARS.

This result therefore summarizes the great potential that DVS cameras offer in the field of low-complexity low-power machine learning algorithms. Along with the results of Section 3.4.4, this justifies the elaboration of an SNN algorithm whose computational complexity is a function of the input sparsity.

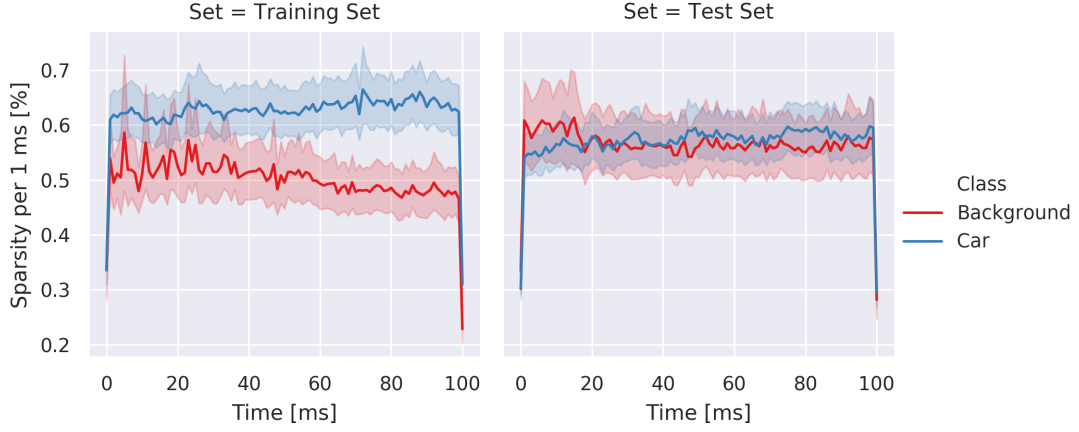


Figure 3.10: Sparsity of the samples for 1 ms time bins vs. Time Stamp in the sample, computed for samples in the $1K$ subset of N-CARS. The thick lines represents the average and the shaded area indicates the standard deviation.

The next section presents the theoretical grounds of the model used in this work to solve the N-CARS classification task. The results found in this section are therefore later used in Section 3.4 when choosing network parameter ranges for hyper-parameter space exploration.

3.2 The Slayer Spiking Neural Network

Throughout this work, the Slayer (Spike LAYer Error Reassignment in Time) SNN [13] model is considered for the following considerations.

- It generalizes the backpropagation algorithm to the SNNs by introducing a substitute derivative for the activation function. As the transformation from the membrane domain to the spiking domain is discontinuous, standard derivation is not feasible and therefore needs to be worked around.
- A loss function based on a target number of spikes is presented.
- It also provides a general formulation based on impulse responses for the neuron and its refractory response. The filters can be adapted so that different models of neurons can be considered.
- It achieves state-of-the-art performance on spiking datasets such as N-MNIST and DVS-Gesture [20].

In the following sections, the forward and backward propagations of the Slayer algorithm are detailed, with an emphasis on the novel substitute derivative.

3.2.1 Formulation

As in any SNN, the information transmitted between layers is a sparse signal of binary values written in Eq. 3.2.

$$s_i^{(l)}(t) = \sum_{f=0}^{S_i^{(l)}} \delta(t - t_{i,f}^{(l)}) \quad (3.2)$$

- $s_i^{(l)}(t)$ is the input spike signal of neuron (i) in layer (l).
- $t_{i,f}^{(l)}$ is the time location of the f^{th} of neuron (i) in layer (l).
- $S_i^{(l)}$ is the total number of spikes of neuron (i) in layer (l).

At the core of the Slayer layer, the *spike response model* (SRM) model [66] rules the transformation from the spiking domain to a continuous *spike response signal* (SRS) by convolving the input spikes with a *spike immediate response kernel*. This is written in Eq. 3.3.

$$a_i^{(l)}(t) = (\epsilon \circledast s_i^{(l)})(t) \quad (3.3)$$

- $a_i^{(l)}(t)$ is the *spike response signal* of neuron (i) in layer (l).
- ϵ is the *spike immediate response kernel*.

The above formulation can optionally include an *axonal delay* as in other SNN models, the spike response kernel is therefore simply shifted by a duration d .

$$\epsilon_d = \epsilon(t - d) \quad \text{where} \quad d \geq 0 \quad (3.4)$$

Similarly, the *refractory response* provides a feedback to the neuron by filtering the output spike signal of the layer with a second filter, as shown in Eq. 3.5.

$$r_i^{(l+1)}(t) = \left(\nu \circledast s_i^{(l+1)} \right) (t) \quad (3.5)$$

- $r_i^{(l+1)}(t)$ is the *refractory response* of neuron (i) in layer (l).
- ν is the *spike refractory response kernel*.

The layer is parametrized by a vector of weights, via a linear function applied on the SRS. The *membrane potential* of the neuron is expressed in Eq. 3.6.

$$u_i^{(l+1)}(t) = f \left(a_i^{(l)}(t), \mathbf{W}^{(l)} \right) + r_i^{(l+1)}(t) \quad (3.6)$$

- $u_i^{(l+1)}(t)$ is the *membrane potential* of neuron (i) in layer (l).
- $f \left(a_i^{(l)}(t), \mathbf{W}^{(l)} \right)$ is later referred as the *Post Synaptic Potential* (PSP) of neuron (i) in layer (l).
- f is a linear function, parametrized by the weight tensor $\mathbf{W}^{(l)}$, performing the core operation of a dense or convolutional layer. Biases are not included in the Slayer model.

Finally, the output spike signal is computed via a *spike function*. Whenever the PSP reaches a predefined threshold θ , a new spike is added in the output spike signal. Thus, the spike function f_s can be formulated as the following :

$$f_s(u) : u \rightarrow s, s(t) := s(t) + \delta(t - t_{f+1}) \quad \text{where} \quad t_{f+1} = \min\{t : u(t) = \theta, t > t_f\}. \quad (3.7)$$

Unlike classical activation functions of ANNs, the spike function's derivative is not defined and therefore raises problems for the back-propagation algorithm.

The authors of Slayer propose spike response kernels of the form

$$\epsilon(t) = \frac{t}{\tau_s} \exp \left(1 - \frac{t}{\tau_s} \right) \Theta(t) \quad \nu(t) = -S\theta \frac{t}{\tau_r} \exp \left(1 - \frac{t}{\tau_r} \right) \Theta(t) \quad (3.8)$$

where $\Theta(t)$ is the step function

$$\Theta(t) = \begin{cases} 1 & \text{if } t \geq 0 \\ 0 & \text{otherwise} \end{cases} . \quad (3.9)$$

The two filters are first order polynomial-exponential products with different scaling parameters ; their functional form is often referred in the literature as *alpha kernels* [18]. As explained further in this section, one property of those filters is of the utmost importance.

The model therefore depends on several hyper-parameters :

- τ_s is the immediate response time constant. Its magnitude affects the span of the filter in the time domain. It is expressed in milliseconds.
- τ_r is the refractory response time constant.
- S is the scale factor of the refractory response. Its magnitude changes the relative importance of the refractory response compared to the immediate response in the PSP equation.

A graph of those filters is shown in Fig. 3.11 for the following parameters, later used in this work.

θ	τ_s	τ_r	S
11.104	7.361	3.067	4.830

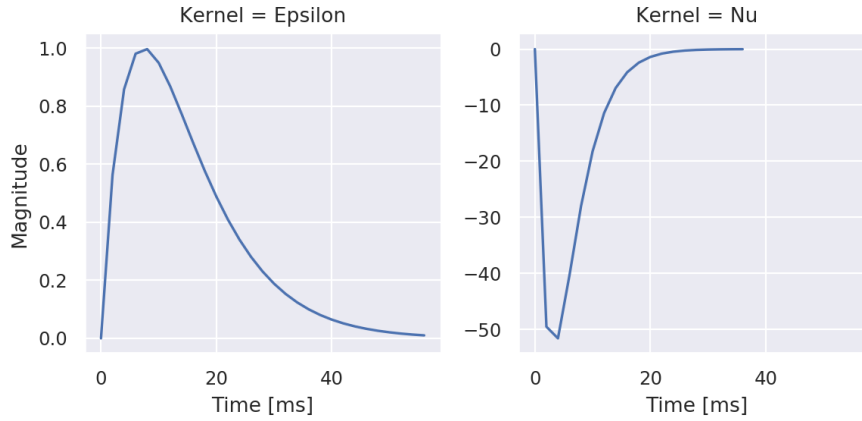


Figure 3.11: Immediate and Refractory kernels

Now that the mathematical foundations of the model are set, the neuron spiking behavior can be analysed. In practice, when operating the model on a computer, the formulation needs to be discretized in the time domain. We can therefore define the sampling as follows :

$$n := \frac{t}{T_s}. \quad (3.10)$$

The variable n is later referred as the *time-step*. The following example of Fig. 3.12 uses a *sampling period* T_s of 2 ms.

1. An input spike of large amplitude is received after 5 time-steps. The amplitude is modified on purpose in this example to account for the impact of the transformation f .
2. The immediate spike response is therefore triggered and crosses the threshold θ after 7 time-steps. At this point, two phenomena occur :
 - An output spike is released at time-step 7.
 - The refractory response is triggered and added to the immediate response to form the membrane potential.

3. Due to the refractory response, the membrane potential dips below the threshold and no additional output spikes are generated. If it were not for the refractory response, output spikes would still be generated since the immediate response is above the threshold.
4. After 30 time-steps, other input spikes enter the neuron but this time with a lower amplitude. Their appearance raises the membrane potential but not sufficiently to cross the threshold so that output spikes are not produced.

The diagram in Fig. 3.13 summarizes the structure of the Slayer neuron and the relations between the different signals. At this point, the feedback loop is clearly visible and could potentially produce an undetermined system when computing the signals. Fortunately, because the refractory kernel of Eq. 3.8 has the property $\nu(0) = 0$, the membrane potential is defined without ambiguity at time-step n ; an output spike triggered at time-step n will only have an impact on time-step $n + 1$ and onwards.

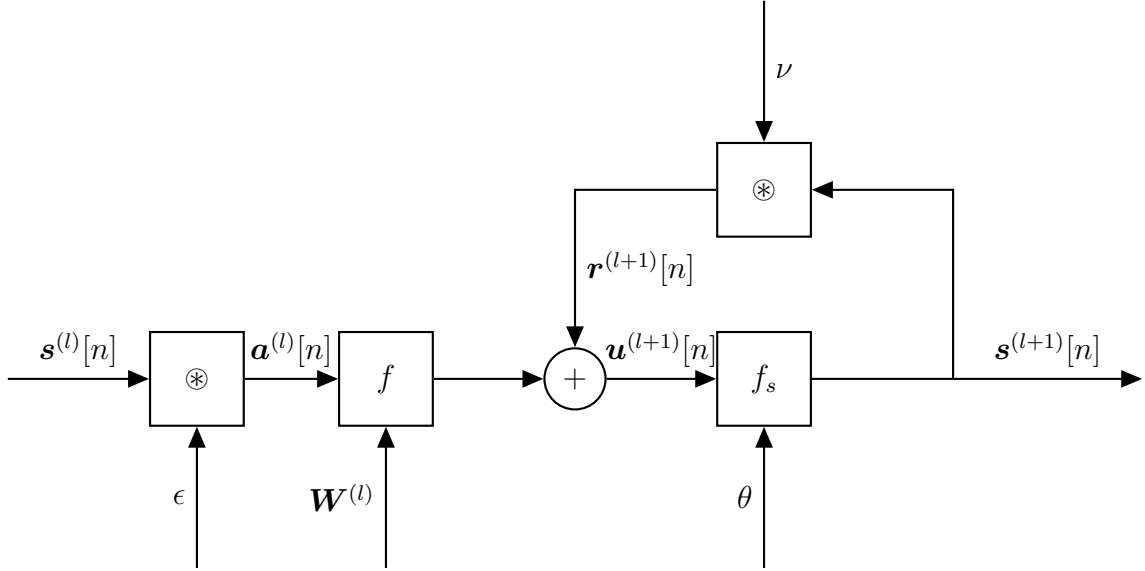


Figure 3.13: Slayer neuron model.

3.2.2 Slayer back-propagation algorithm

The derivations for the back-propagation of the Slayer algorithm are established in this section. The major difference compared to other back-propagation algorithms is located in the expression of the loss function and the derivative of the spike function. These results are the achievement of the authors of the Slayer model [13] ; nevertheless, they are written here with the same systematic approach as in Sections 1.2.4 and 1.3.2. It therefore helps understand the integration of the model in the Pytorch framework in the subsequent chapter.

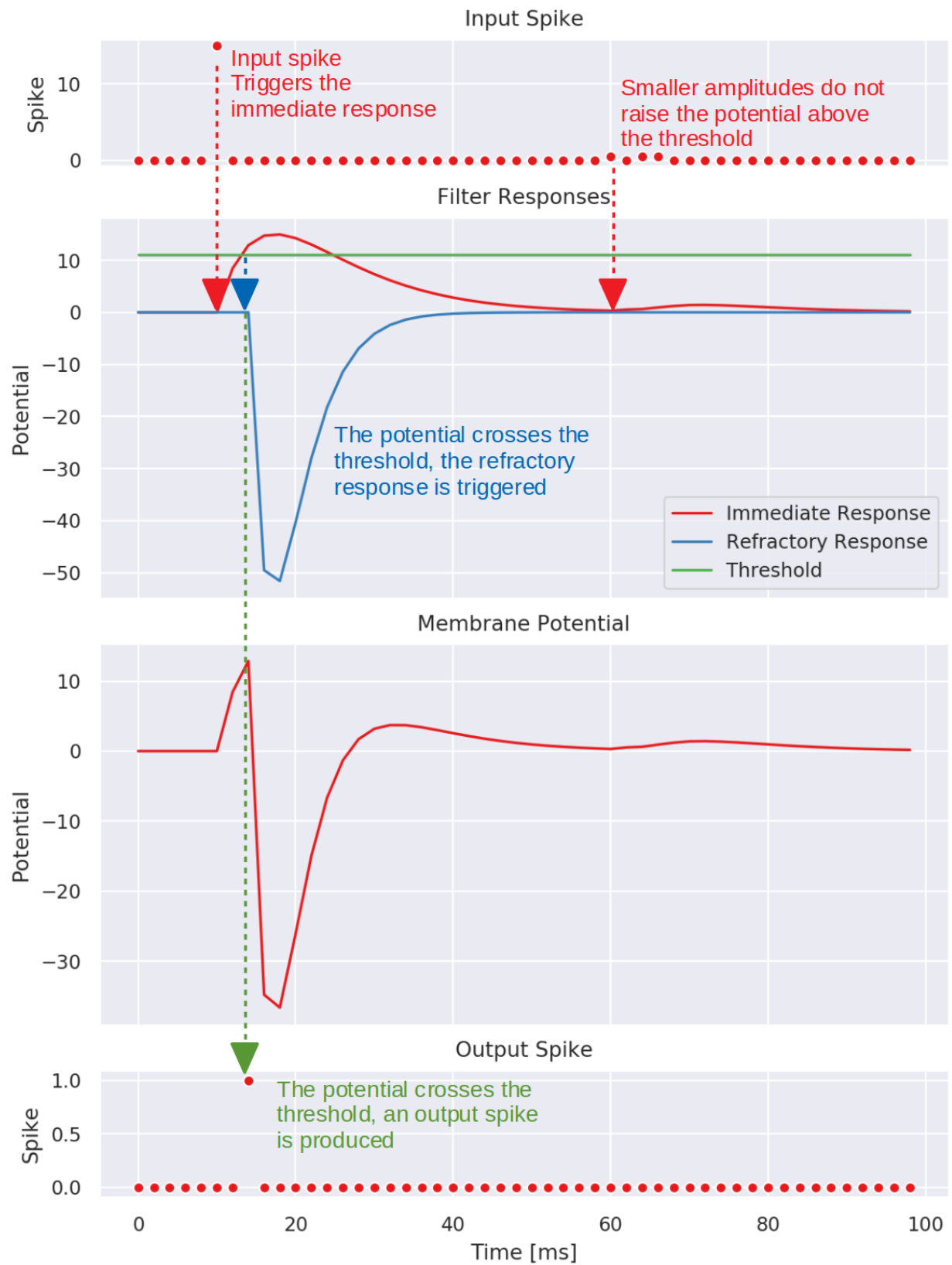


Figure 3.12: Slayer neuron behavior. Top : Input spike signal. Middle-top : Membrane potential decomposed into the immediate and refractory response. Middle-bottom : Membrane potential. Bottom : Output spike signal.

3.2.2.1 Loss function

The introduced loss function has the following form

$$E := \sum_{i=0}^{N^{(l_f)}-1} \int_0^T L(s_i^{(l_f)}(t), \hat{s}_i(t)) dt \quad (3.11)$$

$$E := \sum_{i=0}^{N^{(l_f)}-1} \int_0^T \frac{1}{2} e_i(s_i^{(l_f)}(t), \hat{s}_i(t))^2 dt \quad (3.12)$$

$$= \sum_{i=0}^{N^{(l_f)}-1} \int_0^T \frac{1}{2} (a_i^{(l_f)}(t) - \hat{a}_i(t))^2 dt. \quad (3.13)$$

The square difference between the output SRS of the last layer (l_f) and a target signal is integrated over a window of duration T . The obtained vector is then summed over the layer dimension to obtain a scalar value. The loss function therefore takes into account errors made at any time-step and includes them into the error metric E . As it may be difficult to come up with a target SRS, the following adaptation can be made to imitate the more common choices found in the literature. In classification tasks, the predicted class is in general chosen based on the number of spikes generated in a time frame while the actual location of spikes is irrelevant. The error signal $e_i(t)$ of Eq. 3.12 is replaced by

$$e_i(t) := \int_0^T s_i^{(l_f)}(\tau) d\tau - \hat{S}_i \quad \text{with } t \in [0, T]. \quad (3.14)$$

The target numbers of spike for each output neuron are in the vector $\hat{\mathbf{S}}$ and the error signal has identical values for each t in the duration window. In a binary classification task, the last layer therefore has two neurons, each associated with a class. Two entires are therefore necessary in $\hat{\mathbf{S}}$ and are referred as $\hat{S}_T$ and $\hat{S}_F$ and called the *true* and *false* spike count. The target vector is therefore the following when training the model :

$$\hat{\mathbf{S}}_{\text{class } 0} = \begin{bmatrix} \hat{S}_T \\ \hat{S}_F \end{bmatrix} \quad \hat{\mathbf{S}}_{\text{class } 1} = \begin{bmatrix} \hat{S}_F \\ \hat{S}_T \end{bmatrix}. \quad (3.15)$$

This configuration can be interpreted as a one-hot encoding and can be easily extended to multi-class tasks by only considering one entry with the $\hat{S}_T$ spike count. As explained in the last section, the model is sampled in practice with a time-step T_s .

In order to get rid of the redundant double integration, the spike count is normalized by the number of time-steps N_s . The final loss is therefore

$$\begin{aligned} E &:= \frac{1}{2} \sum_{i=0}^{N^{(l_f)}-1} \left(\sum_{n=0}^{N_s-1} \frac{s_i^{(l_f)}[n] - \hat{S}_i}{N_s} \right)^2 T_s \\ &= \frac{1}{2} \sum_{i=0}^{N^{(l_f)}-1} \left(\sum_{m=0}^{N_s-1} (s_i^{(l_f)}[m] - \hat{S}_i) \right)^2 T_s. \end{aligned} \quad (3.16)$$

The back-propagation algorithm starts by defining the derivative of the loss function with

respect to the output spike signal.

$$\begin{aligned}
ds_i^{(l_f)}[n] &:= \frac{\partial E}{\partial s_i^{(l_f)}[n]} \\
&= e_i \frac{\partial e_i}{\partial s_i^{(l_f)}[n]} T_s \\
\frac{\partial e_i}{\partial s_i^{(l_f)}[n]} &= 1 \\
ds_i^{(l_f)}[n] &= \left(\sum_{m=0}^{N_s-1} (s_i^{(l_f)}[m]) - \hat{S}_i \right) T_s
\end{aligned} \tag{3.17}$$

The gradient of the loss with respect to the output is therefore

$$\mathbf{ds}^{(l_f)}[n] = \left(\sum_{m=0}^{N_s-1} (\mathbf{s}^{(l_f)}[m]) - \hat{\mathbf{S}} \right) T_s. \tag{3.18}$$

Following the diagram in Fig. 3.13, the next operation in back-propagation involves the derivative of the spike function.

3.2.2.2 Spike function

If we consider that the spike function is differentiable, it leads to a path similar to the one we achieved in Section 1.2.4 where the spike function acts as an activation function. This is shown in Eq. 3.19.

$$\mathbf{du}^{(l+1)}[n] = \mathbf{ds}^{(l+1)}[n] \cdot \text{diag}\{f'_s(\mathbf{u}^{(l+1)}[n])\} \tag{3.19}$$

The problem is that the spike function is not differentiable. In fact, at a given time-step n , the spike function acts as

$$s[n] = \begin{cases} 1 & \text{if } u \geq \theta \\ 0 & \text{otherwise} \end{cases}. \tag{3.20}$$

which is the behavior of a step function. In fact, the derivative of a step function is equal to the Dirac delta in the sense of distributions [85].

The subtlety is therefore to find a continuous substitute for the derivative that comprises the correct properties, namely :

- The maximum of the derivative should appear at $u = \theta$.
- The derivative should tend towards 0 as u tends towards $\pm\infty$.

A good candidate is the *spike escape rate function* [86] with an exponential decay. Its formula is written in Eq. 3.21 and its graph for several values of τ_ρ is shown in Fig. 3.14.

$$\rho(u[n]) = \frac{S_\rho}{\tau_\rho} \exp\left(-\frac{|u[n] - \theta|}{\tau_\rho}\right) \tag{3.21}$$

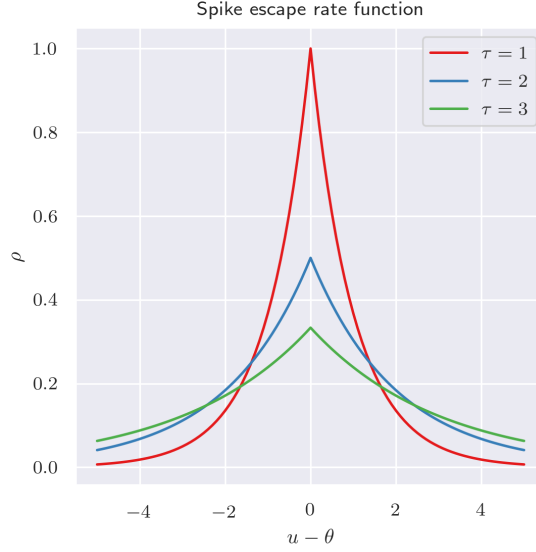


Figure 3.14: Spike escape rate function

The authors of Slayer have further demonstrated that this function can be interpreted as the probability density function (PDF) of the change of state of a neuron in a stochastic version of the model.

3.2.2.3 Layer operation

Pursuing the back-propagation based on Fig. 3.13, the next step is to derive the gradient with respect to the spike response signal $a_i^{(l)}(t)$ and the weights. It therefore depends on the type of operation the layer performs. The main differences with the general back-propagation at this step are twofold :

1. An additional dimension accounting for the time-steps is present in the input and output tensors.
2. As the weights are the same for every time-step, the chain rule also applies on the time dimension. An additional summation on the time dimension is therefore needed.
3. The Slayer model does not incorporate biases in its formulation.

Dense Layer Based on the results of Section 1.2.4, the gradient of the loss with respect to the SRS is given by

$$d\mathbf{a}^{(l)}[n] = d\mathbf{u}^{(l+1)}[n] \cdot \left(\mathbf{W}^{(l)}\right)^T. \quad (3.22)$$

The gradient of the loss with respect to the weights is given by

$$d\mathbf{W}^{(l)} = \sum_{n=0}^{N^{(l)}-1} \left(\mathbf{a}^{(l)}[n]\right)^T \cdot d\mathbf{u}^{(l+1)}[n]. \quad (3.23)$$

Convolutional Layer Similarly, the same calculations can be realized for a convolutional layer. The capital notation is used again to emphasize that the data is arranged in 3D with channel, height and width dimensions and the index notation is defined in Section 1.3. An additional fourth dimension accounts for the time dimension. The convolution and correlation operators therefore act on spatial dimensions. The gradient of the loss with respect to the SRS is given by

$$d\mathbf{A}_{c_i}^{(l)}[n] = \sum_{c_o=0}^{C^{(l+1)}-1} d\mathbf{U}_{c_o}^{(l+1)}[n] \otimes \mathbf{W}_{c_o, c_i}^{(l)}. \quad (3.24)$$

The gradient of the loss with respect to the weights is given by

$$d\mathbf{W}_{c_o, c_i}^{(l)} = \sum_{n=0}^{N^{(l)}-1} d\mathbf{U}_{c_o}^{(l+1)}[n] \otimes \mathbf{A}_{c_i}^{(l)}[n]. \quad (3.25)$$

3.2.2.4 Spike filtering

At last, the back-propagation passes through the temporal convolution with the immediate response kernel. As demonstrated in Section 1.3.2, a convolution operation in a forward pass corresponds to a cross-correlation operation in the backward pass ; the operator acts here on the time dimension. The gradient with respect to the input spike signal is therefore given by

$$d\mathbf{s}^{(l)}[n] = \left(d\mathbf{a}^{(l)} \otimes \epsilon \right) [n]. \quad (3.26)$$

3.3 Pytorch framework

This section presents an overview of the framework used for the training and optimization of the Slayer SNN model on the N-CARS classification task. The authors of Slayer [13] have provided a Pytorch implementation for the new PSP and spiking function blocks as well as adequate I/O functions for preprocessing of the spiking dataset. This work builds on this implementation to provide Slayer with additional tools of deep learning.

1. The N-CARS dataset is interfaced and data augmentation samples are generated based on functions developed from the Albumentations library.
2. A design space exploration of the hyper-parameters of a deep learning model based on Slayer is proposed. Architecture choices such as the number of convolution and dense layers are explored to find a regularized model. Additionally, the hyper-parameters proper to the Slayer neuron formulation are tuned to achieve best performances.

In the end, the characteristics of the optimized model are scrutinized to reveal properties that are at the basis of the Slayer real-time algorithm developed further in this work.

3.3.1 Slayer Pytorch features

All custom modules added to form the Pytorch implementation of Slayer require a set of parameters, located in a configuration `.yaml` file. All parameters are recalled in Table 3.6 and refer to the description of the model in Section 3.2.1.

Conversion	Neuron	Loss
T_{sample} is the time duration of a sample.	θ is the membrane threshold.	$\hat{S}_T$ is the target number of spikes for the correct output neuron.
T_s is the sampling period.	τ_s is the time constant of ϵ filter. τ_r is the time constant of the ν filter. τ_ρ is the time constant of the substitute derivate function. S is the scale constant of the ν filter. S_ρ is the scale constant of the substitute derivative function.	$\hat{S}_F$ is the target number of spikes for the incorrect output neuron.

Table 3.6: Slayer network parameters of the YAML configuration file.

3.3.1.1 AER Conversion to dense tensors

In order to perform the spike filtering and layer operations on the SRS signals, the spikes of the input sample need to be converted from the sparse AER format of the ATIS camera back to a dense tensor. The input tensor is therefore a 4D tensor having the following dimensions :

- The first two dimensions are spatial width and height of the sample. Their sizes correspond to the maximum x and y addresses found in the sample.
- The third dimension is the channel dimension. It has a size of 2 and corresponds to the ON/OFF polarity of the input events.
- The fourth dimension is the time dimension. Its size is equal to

$$N_s = \frac{T_{\text{sample}}}{T_s} \quad (3.27)$$

where T_{sample} is the sample duration and T_s is the sampling period. The time stamps of each spike are therefore rounded to fit inside a time-step in the time dimension.

- An additional fifth dimension may be present if the model processes batches of multiple samples together. For simplicity, this perspective is ignored.

The 4D tensor can therefore be interpreted as a sort of video sequence with only two channels filled with binary values (0-1). The conversion process is illustrated in Fig. 3.15.

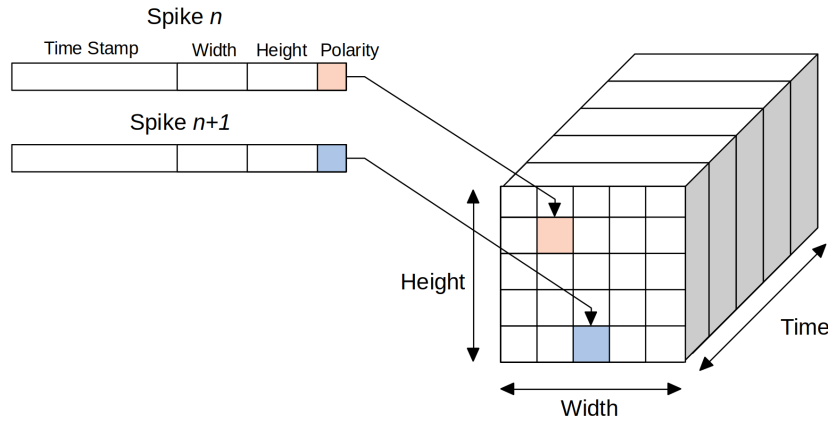


Figure 3.15: Conversion process between the AER format (left) and the dense tensor format (right). The tensor is 4-dimensional, the channel dimension is represented by a color code.

Red: 1, Blue, 0.

3.3.1.2 Module implementation

As explained in Section 1.6.1, Pytorch offers a slick interface to connect research models and benefits from the powerful Autograd back-propagation algorithm. In this way, the Slayer algorithm essentially introduces three new blocks compared to a standard neural network :

1. The SRS is obtained from the input spike signal by a filtering operation in the time dimension with the immediate response filter ϵ .
2. The output spike signal is produced from the PSP by first adding the refractory response ν and then applying the spike function.
3. The MSE loss on the number of spikes is computed by counting the spikes generated from each output neuron and comparing the results to a target count number.

These new blocks are implemented by deriving a Pytorch Autograd Function class, as it was described back in Fig. 1.11.

Spike filtering The **forward** and **backward** methods are implemented with Eq. 3.3 and 3.26 respectively. In practice the convolution or cross-correlation operation with the ϵ filter is implemented as a *Finite Impulse Response* (FIR) filter. The expression of ϵ therefore needs to be truncated as specified in the Python snippet of Fig. 3.16. The refractory response ν is generated with the same method. The filter response is truncated when reaching a value below a given tolerance TOL. It therefore introduces a small deviation compared to the analytical formulation ; this phenomenon is further evaluated in Section 5.1.1.

The 1D convolution and cross-correlation methods in the time dimension are realized with dedicated C++ CUDA kernels, leveraging the power of a GPU to accelerate training.

```

1 def generate_kernel(self, tau, mult = 1, TOL = 0.01):
2     eps = []
3     # t takes values from 0 to (tSample - Ts) with steps of Ts
4     for t in np.arange(0, self.simulation['tSample'], self.simulation['Ts']):
5         epsVal = mult * t / tau * math.exp(1 - t / tau)
6         if abs(epsVal) < TOL and t > tau:
7             break
8         eps.append(epsVal)
9     return eps

```

Figure 3.16: FIR generation method for the ϵ and ν filters in the Pytorch framework. `tSample` indicates the total time duration of the sample and `Ts` indicates the sampling period.

Spike function As in the spike filtering Autograd function, the methods for forward and backward passes are implemented using Eq. 3.6 and 3.19. In the forward pass, the FIR filter of the refractory response is added to the PSP once it has reached the threshold θ and a spike is added at the corresponding time index in the output tensor. For the backward pass, the developments of Eq. 3.19 indicates the gradient to compute. It involves the substitute function in Eq. 3.21.

Loss function At last, the loss function is implemented with a Pytorch Module. This class only require the specification of the forward method and is implemented with Eq. 3.16. The backward method is automatically derived from the tensor operations of the forward method.

3.3.1.3 Adaptation of the Pytorch dense and convolutional layers

Due to the additional time dimension in the processed tensors, the convolution and dense operations need to be adapted. They are effectively interfaced with the `Conv3D` module of Pytorch that performs a convolution operation in a 3-dimensional space. The interface is performed in the following manner :

- For a 2D convolutional layers with input and output channels $C^{(l)}$, $C^{(l+1)}$ and filters of size $K_h \times K_w$, the corresponding 3D convolutional layer has the same input and output channels parameters and has filters of size $K_h \times K_w \times 1$. Hence, there is effectively no convolution operation in the time dimension, as illustrated in Fig. 3.17.
- For a dense layers with input and output channels $N^{(l)}$, $N^{(l+1)}$, the corresponding 3D convolutional layer has the same input and output channel parameters and has filters of size $1 \times 1 \times 1$.

The bias is disabled by default for both dense and convolutional layers. The 3D convolution operation automatically performs the gradient computation in Eq. 3.25, with a sum on the third dimension.

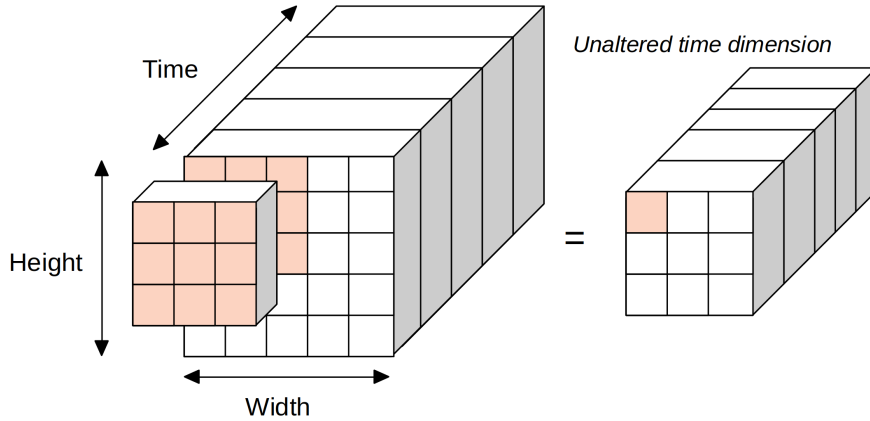


Figure 3.17: 3D Convolution operation with a filter of size $3 \times 3 \times 1$, leaving the third dimension's size unaffected.

Concerning max-pooling, the Slayer model opts for the following formulation. The spikes in $K \times K$ windows are summed together and multiplied by a constant weight value greater than the threshold. In this way, if any spike is present in the window, it is transmitted to the output spike train of the pooling layer.

3.3.2 Pytorch Lightning Framework

The Pytorch Lightning framework is used in this work in order enhance the original Pytorch implementation of Slayer, in the hope to provide competitive performances on the N-CARS dataset.

Automatic model generation Starting from the YAML configuration file already present in the program, parameters are added to entirely define a model’s architecture and training parameters from the information contained in the file. An example of such configuration is displayed in Appendix A.1 for the baseline model of Section 3.4.1. The additional parameters are summarized in Table 3.7.

Architecture	Dataset	Training
• Ordered list of convolutional and dense layers with their related parameters.	• Input image dimensions • Training / validation set relative percentages • Batch size • Dataset path	• Optimizer name and related parameters • Learning rate scheduler and related parameters • Maximal number of epochs • Early stopping configuration

Table 3.7: Added parameters in the YAML configuration file for automated model generation.

When instantiating the model before the training phase, the YAML configuration is loaded and automatically generates the corresponding network. This step is essential in order to later perform a hyper-parameter design space exploration, where hundreds of models are generated with different parameters.

Dataset preprocessing Pytorch Lightning allows to automate the dataset preprocessing by providing training / validation splitting functions, shuffled batch fetching methods and the possibility to include data augmentation transformations on the data. Among the multiple evaluation methods that exist, a simple hold-out validation split is chosen in order to limit the training duration per model. The separate test set is chosen as the one already provided by the authors of the N-CARS dataset. The image transformation that performed on the input samples are detailed below in Section 3.3.3.

Callbacks The performance metrics of the model are measured at every epoch on the training and validation sets in order to detect overfitting and trigger an early stopping callback. This callback is configured with a *patience* parameter that sets the maximum number of epochs that can elapse without an increase of the model’s performances ; when the callback is triggered, the model state is restored at the epoch the model achieved its best metrics on the validation set. The metrics are also transmitted to an online visualization tool for monitoring the evolution of the learning. This is especially useful when conducting the hyper-parameter described in Section 3.3.4.

Metrics On the training and validation set, the accuracy, precision, recall and loss are recorded at each epoch. The accuracy metric is selected as the metric to maximize since the N-CARS dataset is a binary classification problem that has nearly equally populated classes (51.33 / 48.67 %). Nevertheless, the precision and recall metrics are recorded to provide additional information about the miss-predicted samples.

3.3.3 Data augmentation

As seen in Section 1.5, data augmentation is a powerful technique to limit overfitting in CNN models. However, the N-CARS dataset has a particular format and cannot just admit any transformation. Once a fetched sample has been transformed into a 4D tensor, it contains exclusively binary values. However, an image transformation such as rotation or a scaling involves a form of resampling of the original pixels to form the output image and most of the time an averaging or *bilinear* interpolation technique is used to compute the value of new pixels. In this instance, a *nearest* interpolation must be used in order to avoid averaging and keep a binary output. Indeed, the pixels of the output image are computed by selecting the nearest corresponding pixel when applying the inverse mapping between the output and input coordinates. This can be implemented using the Albumentation library [43], as illustrated in Fig. 3.18 on a sample of the MNIST dataset.

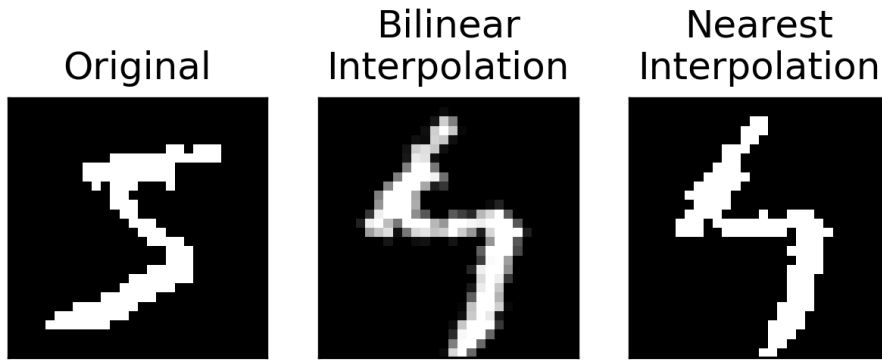


Figure 3.18: Interpolation method comparison on a sample of the MNIST dataset.

Once this has been established, a second problem arises. The random transformation associated with the data augmentation must be identically applied to all time-steps of the 4D input tensor. The Albumentation library does not propose such augmentations on video data. Therefore an extension of the library was written to deal with this operation. Finally, the library allows to chain efficiently multiple transformations related to different needs. Indeed, before augmentation, the input images needs to be resized to a fixed size in order to be processed by a CNN. In conclusion, the dataset samples are modified as follows :

- The training and validation set images undergo the following sequence of transformations
 1. Resizing of the images into a fixed size array.
 2. Data augmentation with random rotation, shifting and scaling transformations.
 3. Conversion to a Pytorch Tensor.
- The testing set images are processed with the following sequence :
 1. Resizing of the images into a fixed size array.
 2. Conversion to a Pytorch Tensor.

The parameters corresponding to the resizing and the random transformations are specified in Section 3.4 when conducting the hyper-parameter search.

3.3.4 Random sampling and experiments

At last, the Optuna library [44] is used to create a wrapper around the pytorch model and perform hyper-parameter space exploration. The model is trained inside an `objective` method that returns the final validation accuracy of the trained model. Hyper-parameters inside the objective function are selected from defined distributions such as :

- An uniform distribution between two bounds.
- A log-uniform distribution between two bounds.
- A categorical uniform distribution across a list of discrete values.

A `study` object iteratively calls the `objective` function and feeds it with sampled hyper-parameters. The sampling process is also customizable. A random sampler will draw independent identically distributed hyper-parameters from the specified distributions while other sampler can leverage the acquired knowledge of the design space to concentrate the sampling in a specific region. Thanks to the upgraded YAML file, the `objective` method easily write a sampled architecture into the configuration file before instantiating the model.

Finally, Optuna offers a callback function compatible with Pytorch Lightning that allows to *prune* a model before completion of its training phase, based on the collected statistics about the validation accuracy of the past models. This allows to save training time by aborting the training of models that are doomed to fail. Pruners include the following algorithms :

- *Percentile pruners* regularly check if the trained model’s metric is within the bottom x -percentile of the accumulated distribution of past model metrics and stops its training if it is the case.
- *Threshold pruners* stop the training of a model if its metric falls below a certain value at a given epoch.

3.4 Experiments

Based on the Pytorch Lightning framework presented above, a series of experiments are conducted in order to find an optimized model to solve the N-CARS task. First a baseline model is created in order to establish an accuracy metric to beat and assess the performances of Slayer with unoptimized parameters. Then, an hyper-parameter parameter search is conducted in a two-step approach :

1. The neuron parameters proper to the Slayer formulation are set to default values while different architectures and learning rates are tested.
2. Once a suitable choice of architecture is selected, the neuron parameters are modified.

This approach is known to be sub-optimal as the two steps momentarily freeze several hyper-parameters while optimizing others. Moreover, the range of tested parameters is restricted to predefined bounds. On the other hand, it provides a reduced complexity for the search and separates the parameters in a sensible way. A few additional hypotheses are made concerning the architectural choices that remain fixed for both optimization steps :

Dataset The optimization is conducted on the $1K$ subset established in Section 3.1. This is necessary in order to reduce the execution time of an epoch and allows to test several models in a reasonable time. The main modification is that in this subset the two classes are equiprobable, which differs slightly with the overall dataset, and the overfitting is likely to occur faster due to the limited number of samples. This subset is divided into a training and a hold-out validation set with a 80/20 % random split.

Image resizing The samples need to be resized into a fixed image size in order to be processed by the model. Two options are available :

1. Resizing the images into a fixed size using *nearest* interpolation.
2. Cropping large images, discarding a part of the image and padding small images with a constant values to increase its size.

As explained in Section 3.1, the N-CARS dataset was built using a region-based detection algorithm and the produced regions were manually adjusted to perfectly contain the objects of interest (Fig. 3.1). Therefore, a cropping method would likely eliminate informative parts of the image and would result in a loss of accuracy, especially on the *Car* class. Cropping can be avoided if the images are resized to the maximum size of 120×100 (width $\times$ height) since only padding would be used. However, images of this size are very large and require a model with a high computational complexity ; for comparison, the popular CIFAR-10 dataset only contain images of size 32×32 . The remaining solution is therefore to resize large images into a fixed size with a nearest interpolation method. Small images could still use padding to fit in the fixed size image but this option is discarded as all images are likely to contain a car or background element in its entirety. Indeed, a padding would result in objects of variable sizes in the dataset, which results in patterns that are more complex to learn.

The selected fixed size is 53×41 , which is slightly under the average sizes of all images, as was shown in Table 3.2. The data of the training and validation are then augmented with a series of random transformations in order to reduce overfitting. The considered transformations are :

- Random height and width shifting by an amount between -10 and 10 %.
- Random scaling by a factor ± 10 %.
- Random rotations with an angle between -15 and 15 %.

The missing pixels are replaced with a 0. An example of data augmentation on a *Car* sample is shown in Fig. 3.19.

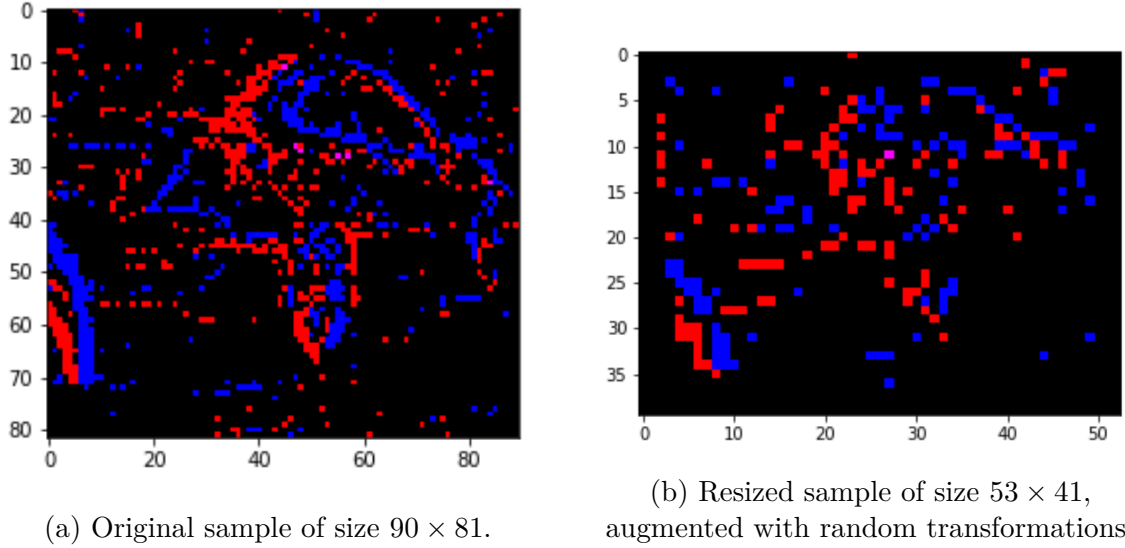


Figure 3.19: Data augmentation example of *Car* sample of the N-CARS dataset. Refer to Fig. 3.2 for the color annotations.

Down-sampling and filter size Convolutional layers usually require a down-sampling operation in-between layers in order to limit the number of neuron and learn hierarchical patterns. The max-pooling operation in the Slayer formulation requires additional computations to derive a membrane potential and output spikes, which significantly lengthen the execution time as the number of neurons in these layers is quite large. We therefore opted for a down-sampling using strided convolutions. With stride size of 2 is selected to reduce the feature map sizes by a factor of 2 while also lowering the complexity of the convolution operation.

The filters used for the convolutional layers have a fixed 5×5 format. Indeed, input sample and intermediate feature maps keep a large spatial size and the features to learn are expected to be spread in the image. A large filter therefore size provides a large patch to detect patterns.

At last, only *valid* convolution operations are considered in order to further limit the number of neurons present in each layer. The spatial size of the feature maps after each convolution is therefore dictated by Eq. 3.28.

$$N^{(l+1)} = \frac{N^{(l)} - K}{2} + 1 \quad (3.28)$$

The choice of the input image format was also dictated by this equation. The size was adjusted in order to support up to 3 convolutional layers without the need of truncating the intermediate feature maps. Indeed, if the quantity $N^{(l)} - K$ is odd, the output size is floored to the nearest integer and the down-sampling has a factor greater than 2. The input image size is therefore chosen to avoid such cases.

Optimizer and training parameters The selected optimizer for the back-propagation algorithm is the Adam optimizer [29]. This algorithm is a first-order gradient-based optimization that is computationally efficient with little memory requirements. Furthermore, its authors qualify it as well suited for problems that are large in terms of data and/or

parameters. Additionally, the AMSGrad variant of Adam is used [87] ; it solves known convergence pitfalls of Adam and has shown to provide better performances on CNN models. The batch size is fixed to 32 samples due to memory constraints of GPU used for the training.

3.4.1 Baseline model

As explained in Section 3.1, the statistical baseline to beat is the 51.33 % accuracy obtained with a random guessing. The following model therefore presents a simple architecture with parameters detailed in Table 3.8 and 3.8.

Layer	Hyper-parameter	Value
Conv. 1	In. Channels	2
	Out Channels	16
Conv. 2	In. Channels	16
	Out Channels	16
Dense 1	In. Channels	$11 \cdot 8 \cdot 16$
	Out Channels	2
	LR	0.01

Table 3.8: Baseline model architecture parameters. LR refers to the learning rate of the optimizer.

Simulation		Neuron						Loss	
T_s [ms]	T_{sample} [ms]	θ	τ_s	τ_r	τ_ρ	S	S_ρ	$\hat{S}_T$	$\hat{S}_F$
1	100	10	10	1	1	2	1	90	10

Table 3.9: Baseline model neuron parameters.

The neuron parameters are selected from the test examples given by the authors of Slayer in their Pytorch implementation for the MNIST dataset and provide a good starting point for further optimization. The model therefore contains two convolutional layers and a single dense layer for the binary classification. Due to the strided convolutional layers with a kernel filter of size 5, the number of neurons before the dense layer is equal to $11 \cdot 8 \cdot 16 = 1408$. For this first test only, the data augmentation is disabled and the model is trained on the complete N-CARS dataset, in order to measure the effect of overfitting. The learning curves associated with the training phase is displayed in Fig. 3.20. The training lasted for 26 epochs and was interrupted by the Early stopping callback, as no significant improvement on the validation accuracy was observed during the last 6 epochs.

The complete model’s metrics are displayed in Table 3.10. It shows already considerable performances with a validation accuracy just under 90 %. Moreover, the precision and recall metrics indicate that the errors are balanced between false positives and false negatives. There is a loss of around 8 % of accuracy between the training and testing set, indicating a slight overfitting.

This model therefore proves that Slayer is able to achieve competitive performances on the N-CARS dataset with a small CNN model. In addition, it does not invalidate the fixed parameters choices made at the beginning of the section, concerning the image resizing and

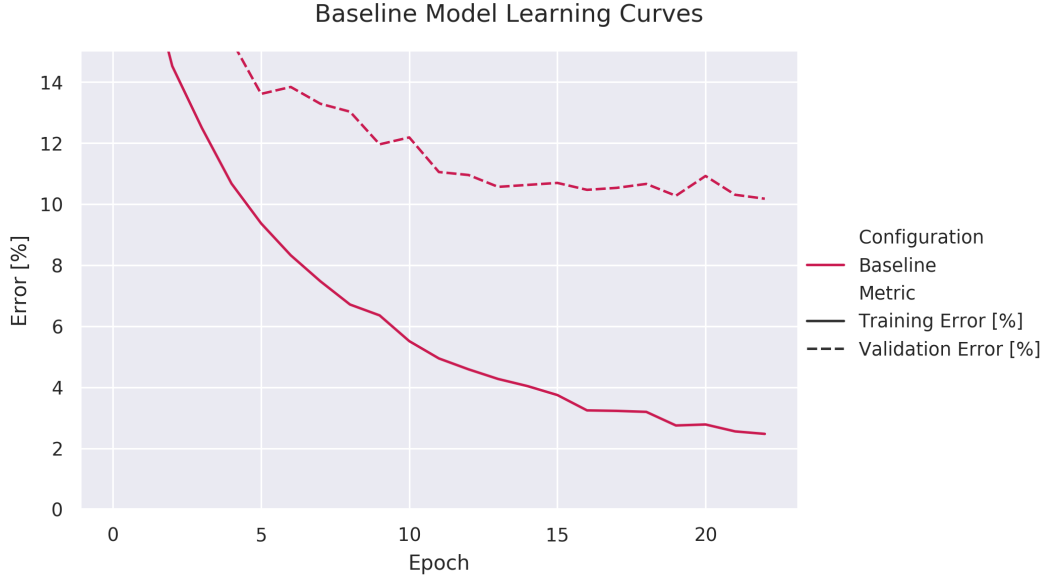


Figure 3.20: Training curves for the baseline model with hyper-parameters of Table 3.8 and 3.9. The error metric in percent is measured as $\text{Error} = 100 - \text{Accuracy} [\%]$.

Set	Error [%]	Accuracy [%]	Precision	Recall	Loss
Training	2.48	97.52	0.9745	0.9777	97.496
Validation	10.18	89.82	0.8918	0.9132	197.253

Table 3.10: Baseline model metrics

stride size. The next step involves exploring the design space in order to increase the accuracy, with the help of data augmentation techniques and random sampling of hyper-parameters.

3.4.2 Architecture space exploration

This first design exploration involves tuning various parameters related to the architecture of the model. Based on the values of the baseline model, the parameters and their respective distributions are selected in Table 3.11. On the other hand, the parameters related to the Slayer neuron are kept at their default value of Table 3.9.

Hyper-parameter	Distribution
Conv. Layers	Cat-uniform[2,3]
Channels Conv.	Log-uniform[8,64]
Dense Layers	Cat-uniform[0,1]
Channels Dense	Log-uniform[64,256]
LR	Log-uniform[0.001,0.1]

Table 3.11: Architecture design space exploration hyper-parameter distributions.

The number of convolutional layer is chosen at random between 2 and 3. Identically, the number of hidden dense layers is picked at random between 0 and 1 layer. The number of channels in each layer is sampled from a log-uniform distribution. Note that the last dense

layer always as a number of channels equal to 2 in order to output the prediction in a binary form. The learning rate LR is also sampled from a log-uniform distribution.

The design space exploration is configured with a random sampler that draw model hyper-parameters without taking into account the performance of past trained models. In order to promote models with good performance during the search, a pruner is configured to stop the training of models that fall into the bottom 25 % percentile (last quartile) of the accumulated validation accuracy distribution. The pruning process starts after the training of 5 models in order to build accurate statistics on the validation accuracy. Models are given 10 epochs of warmup before being compared every 5 epochs with the validation accuracy distribution. Based on the results of the baseline training, the models are given maximum number of 50 epochs for training.

The results are shown in Fig. 3.21, the different graphs expose the relations between every pair of hyper-parameter with respect to the recorded validation accuracy. Some of the graphs present interesting information regarding a sensible choice of hyper-parameter. These graphs are commented in the paragraph below and addressed by their (x, y) coordinates in the plot.

- Models that include 3 layers of convolution perform very poorly, with a validation accuracy barely above a random guessing (graphs (1,6)). Indeed, the kernel size of the last layer is too large compared to the input feature map size.
- The convolutional layers with a number of channel below 40 generally perform worse than the others (graph (2,4)). Preferably, the number of channels should be similar between the two convolutional layers.
- The models show best performances when 1 hidden dense layer is added between the convolutional layers and the output dense layer (graph (5,1)).
- The number of neurons in the hidden dense layer should stay around 128.
- The ideal learning rate is around the value 0.01.

Based on those results, the architectural parameters of Table 3.12 are selected. The number of neurons in the dense layers and the learning rate are still left undetermined and will provide additional degrees of freedom in the next test.

Layer	Hyper-parameter	Value
Conv. 1	In. Channels	2
	Out Channels	64
Conv. 2	In. Channels	64
	Out Channels	64
Dense 1	In. Channels	$11 \cdot 8 \cdot 64$
	Out Channels	N.A.
Dense 1	In. Channels	N.A.
	Out Channels	2

Table 3.12: Selected architecture hyper-parameters. The convolution layers have filters of size 5×5 and a stride size of 2.

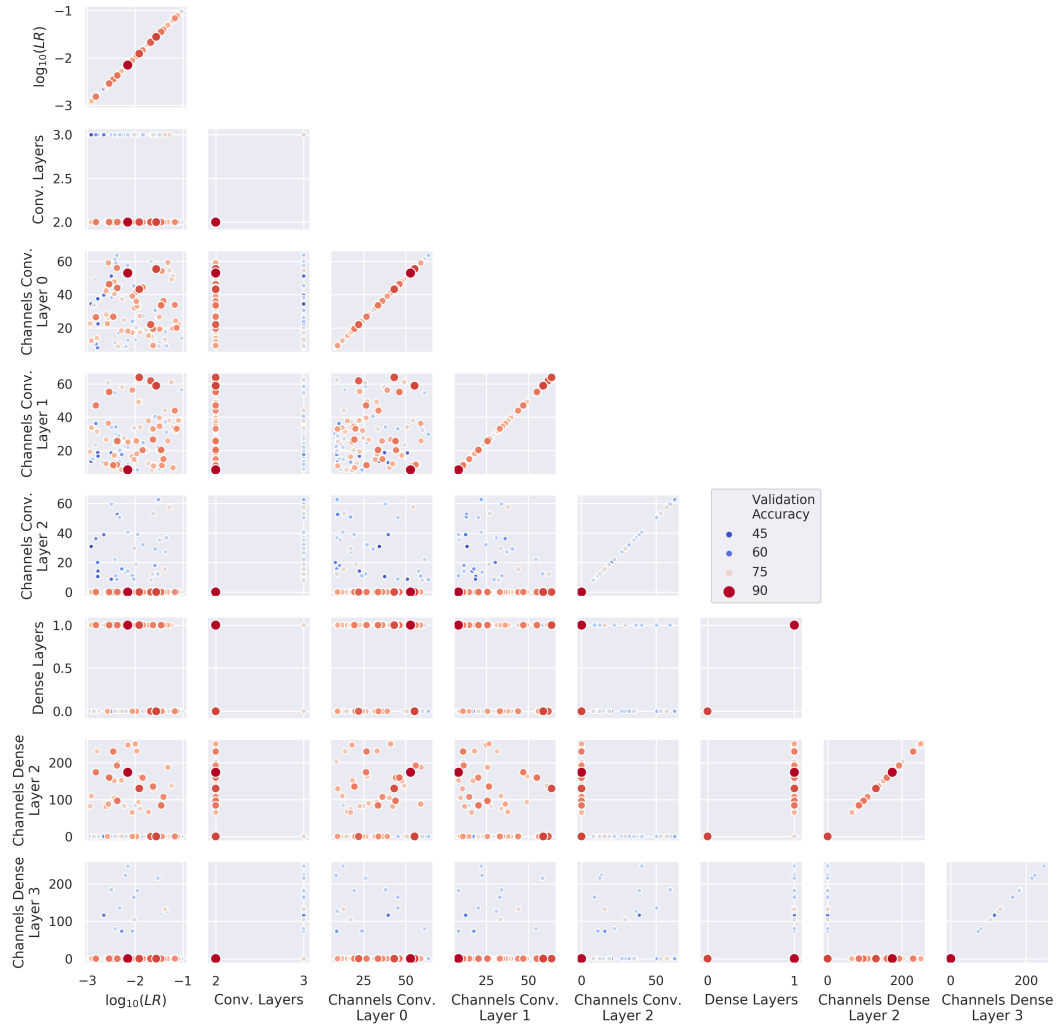


Figure 3.21: Architecture hyper-parameter design space exploration.

3.4.3 Neural parameter space exploration

This second hyper-parameter optimization mainly focuses on the parameters related with the Slayer model. The distributions of the tuned parameters are specified in Table 3.13.

Hyper-parameter	Distribution
T_s [ms]	Cat-uniform[0,1]
T_{sample}	100
θ	Uniform[1,50]
τ_s	Uniform[2,20]
τ_r	Uniform[0.5, $\min(\tau_s, 5)$]
τ_ρ	Uniform[1,10]
S	Uniform[1,5]
S_ρ	1
$\hat{S}_T$	Uniform[$1.1N_s/2$, $0.9N_s$]
$\hat{S}_F$	Uniform[$0.1N_s$, $0.9N_s/2$]

Table 3.13: Neuron design space exploration hyper-parameter distributions.

Some parameters have conditional distributions based on the value of other parameters. Namely, τ_r is required to remain below the value of τ_s in order for the refractory response to have a shorter effect than the immediate response. The spike count $\hat{S}_T$ is required to always remain above the spike count $\hat{S}_F$ and their values depend on the number of time-steps in the model (Eq. 3.27). Note that the values remain within a 10 % margin between the minimal and maximal number of spikes.

The sample length parameter T_{sample} is kept unchanged as Section 3.1 has shown that events were equally distributed across the sample duration. Also, the scale parameter of the substitute derivative S_ρ is not modified. At last, the number of neurons in the hidden dense layer and the learning rate keep the same distribution as in the previous experiment. The results of this experiment are shown in Fig. 3.22 and indicates the following trends :

- The main result obtained in this experiment is that models with great performances can be obtained with a sampling period equal to 2 ms (graph (2,7)). Furthermore, this configuration allows to considerably lower the execution time of the forward and backward passes in the network because of the smaller processed tensors. This observation plays a main role when optimizing the model for real-time performance in Chapter 5.
- The model is able to learn correctly with relatively low output firing rates (graph (3,5)). In addition, the separation between the correct and incorrect spike count does not need to be maximal.
- The threshold θ should keep a value around 10. A high threshold yields poor performances (graph (5,4)).
- The scale parameter S of the refractory response should have values around 5 (graph (6,3)).
- The same conclusions regarding the learning rate and the number of neurons in the dense layer can be drawn from this experiment.

- Surprisingly, the value of τ_ρ has very little influence on the performance (graph (9,0)) although the spike function's derivative is a central part of the Slayer formulation.
- Setting apart the influence of the learning rate and the threshold, all neuron hyper-parameters show a remarkable robustness and their variations do not induce observable changes in accuracy. This probably indicates that most of the information required for a correct classification is contained in the spatial domain and not the time domain.

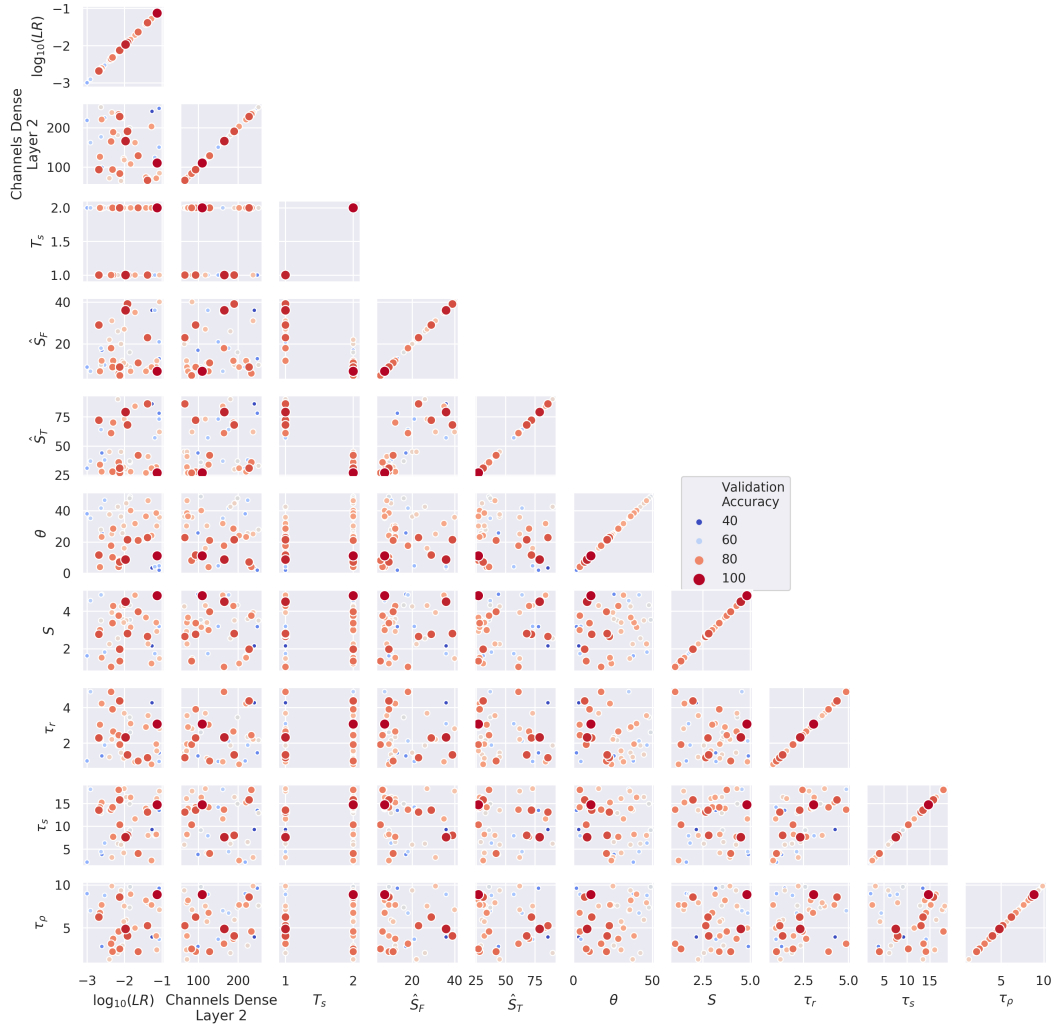


Figure 3.22: Neuronal hyper-parameter design space exploration.

Among the various models created in this design space exploration, the one that achieved the best validation accuracy is selected for the next step of the optimization process. The model needs to be trained on the entire dataset and the learning rate can be adjusted one last time to give best performances. The neuron parameters of the best selected model are shown in Table 3.14 ; its architecture was already selected in Table 3.12 and its YAML configuration file is shown in Appendix A.2.

Hyper-parameter	Value
T_s [ms]	2
T_{sample}	100
θ	11.104
τ_s	7.361
τ_r	3.067
τ_ρ	8.873
S	4.828
S_ρ	1
$\hat{S}_T$	27
$\hat{S}_F$	7
Dense 1 Out Channels	128

Table 3.14: Neuron hyper-parameters of the selected model of the hyper-parameter search.

3.4.4 Optimized model

3.4.4.1 Performance metrics

When training the model on the complete dataset a scheduler is added to the optimizer to reduce the learning rate at specific epochs. This technique accelerates the training and provides an increase in performance. The scheduler multiplies the learning rate by a factor $\gamma < 1$ and effectively reduces the step size of the gradient descent, allowing to better approach a minimum of the loss function. The scheduler parameters are shown in Table 3.15 while the learning curves are displayed in Fig. 3.23. A steep decrease in the training error is visible at epoch 15 due to the action of the learning rate scheduler.

Hyper-parameter	Value
LR	0.0743
γ	0.1
Epochs	[15, 25]

Table 3.15: Scheduler parameters for the optimized model. The initial learning rate LR is multiplied by γ at two specific epochs.

The final statistics are computed for the training, validation and test sets in Table 3.16. An increase of 4.7 % was achieved compared to the baseline model. The training and validation metrics show a very limited overfitting and balanced performances for the precision and recall metrics. On the other hand, the metrics on the separate test set show a considerable drop of 6 % of accuracy. This could be the result of two phenomena :

1. Due to the intensive hyper-parameter optimization, we may have reached overfitting on the validation set.
2. The differences highlighted in the N-CARS dataset analysis of Section 3.1 could have an influence on the model's metrics. Indeed, the distributions of image sizes and number of events showed noticeable differences between the training and test sets established by the authors of N-CARS.

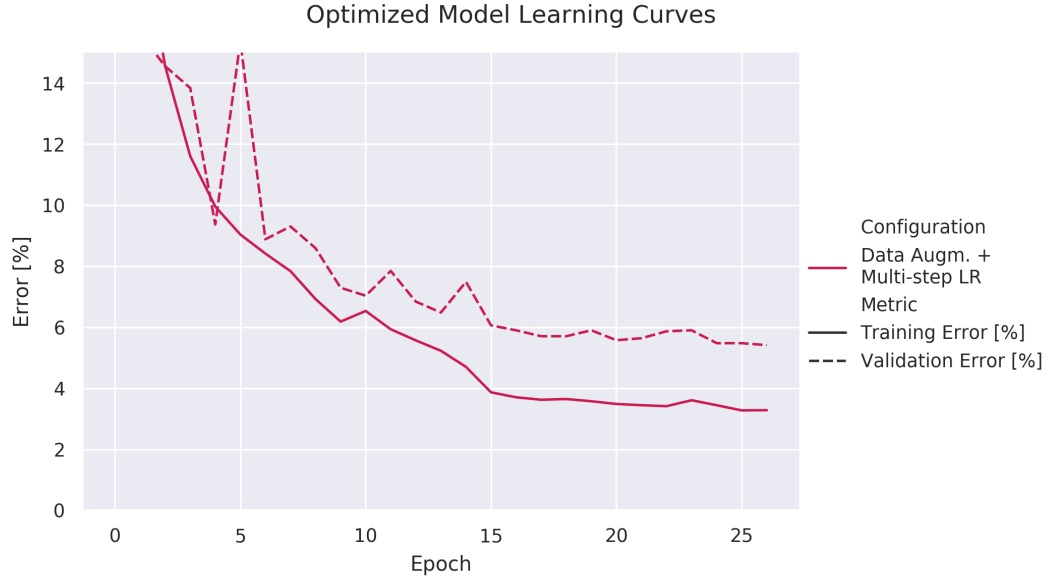


Figure 3.23: Training curves for the optimized model with hyper-parameters of Table 3.12, 3.14 and 3.15. The error metric in percent is measured as $\text{Error} = 100 - \text{Accuracy} [\%]$.

Set	Error [%]	Accuracy [%]	Precision	Recall	Loss
Training	3.28	96.72	0.9636	0.9736	17.173
Validation	5.41	94.59	0.9330	0.9596	25.377
Test	11.41	88.59	0.8922	0.8813	46.076

Table 3.16: Optimized model metrics.

3.4.4.2 Discussion

In comparison to state-of-the-art accuracies achieved on the N-CARS dataset (Table 3.17), our model reaches performances 1.61 % under the accuracy of the HATS model [17]. It is also outperformed by cloud-computing class algorithms [77] [18] that reaches 91.0 % and 92.5 % accuracies but remains with a complexity orders of magnitude lower than power-hungry ResNet models.

Regarding the performances achieved by Prophesee (95.5 %), their results correspond to our accuracy on the validation set. Their architecture is reported to be composed of 2 convolutional layers with 20 and 50 channels respectively, followed by 2 dense layers of 500 and 2 neurons [65]. Their solution might therefore yield a better trade-off between the accuracy and the model complexity ; unfortunately, their experimental setup is not reported for fair comparison.

Dataset	Method	Accuracy
N-CARS	HOTS [70]	62.4 %
	Gabor-SNN [17]	78.9 %
	Grid-search AIF [65]	81.97 %
	Our Work	88.59 %
	HATS [17]	90.2 %
	Rebecq et al. [77]	91.0%
	EST [18]	92.5 %
	Prophesee	95.5 %

Table 3.17: Reported accuracies for the N-CARS dataset.

3.4.5 Model statistics

To conclude this analysis, a few statistics of the optimized model operation are drawn in this section. First, the model’s weights distributions are drawn in Fig. 3.24 for each layer. The weights of the convolutional layers have values ranging from -50 to 20, with a negative mean value. The distributions of the weights in the dense layers are more concentrated around 0, with values ranging between -5 and 5. This observation is rather interesting, most input spikes are therefore convoluted with negative weights. Since the threshold value that triggers output spikes is positive, it means that most patterns detected by the filters are will push the membrane potential in the negative values in order to avoid the generation of an output spike. Only a few patterns will produce positive membrane values that crosses the firing threshold. Indeed, the graphs of Fig. 3.25 and 3.26 show the distributions of the number of output spikes generated by each layer for each time-step when processing the samples of the N-CARS test set. The number of spikes is normalized by the number of neurons in the layer in Fig. 3.26 in order to represent the quantity as a sparsity percentage. Several interesting points are raised :

- The distribution of events for the *Input* layer, corresponds to the recorded sparsity of the input samples and it is similar to the one of Fig. 3.9 already presented in Section 3.1. The original samples were showing sparsity below 1 % while this distribution

shows sparsity values generally below 2 %. This doubling is due to the sampling period considered ; in this case 2 ms time-steps are considered while the graph presented in the N-CARS dataset analysis was for 1 ms time-steps. This effectively shows that the image resizing did not fundamentally change the sparsity distribution of the input samples as the main differences are explained by the change of sampling period.

- The first convolutional layer presents sparsity values below 0.5 %. As predicted by the weight distribution analysis, only a very restricted number of events is produced in this layer. We can observe on an average a slightly higher number of events produced for the *Car* distribution.
- The sparsity rises for the output produced by the second convolutional layer but remains in a range below 5 % most of the time. As in the first layer, the *Car* class produces more input events than the *Background* class. Even though the input samples have a slightly different average number of events when entering the model, this difference is increased in the convolutional layers. It therefore means that the positive filter weights probably detect car patterns in the feature maps.
- The dense layers contain a greater proportion of spiking neurons than the convolutional ones, probably because they detect global patterns across their input. When looking at the absolute number of events in the distributions of Fig. 3.25, the second convolutional layer comprises the largest amount of events per time-step.

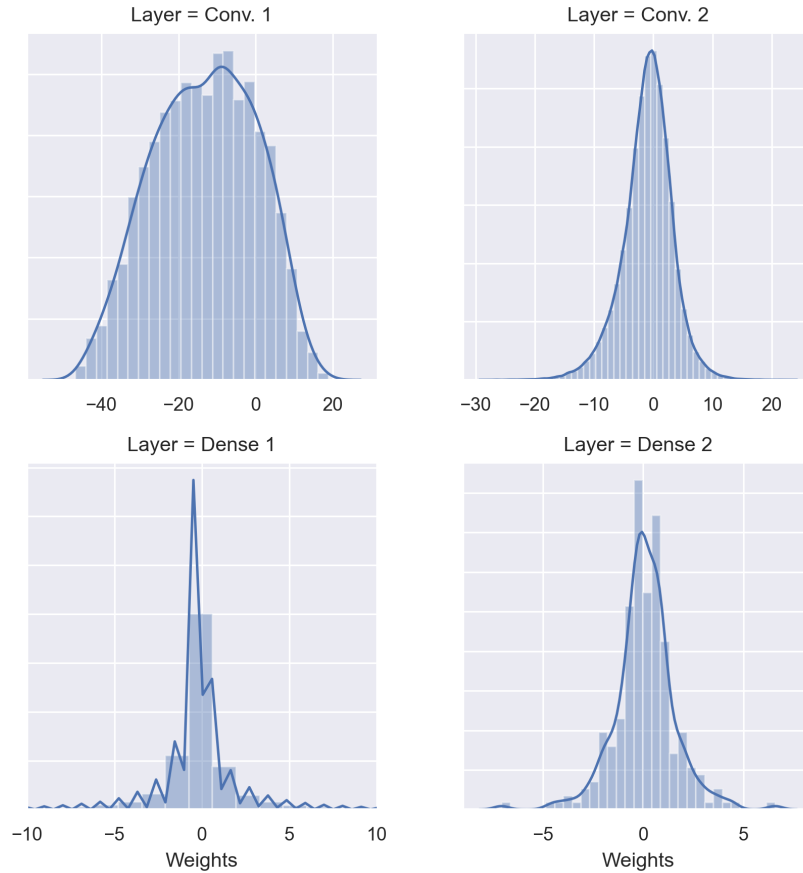


Figure 3.24: Distribution of the weights for each layer of the optimized model.

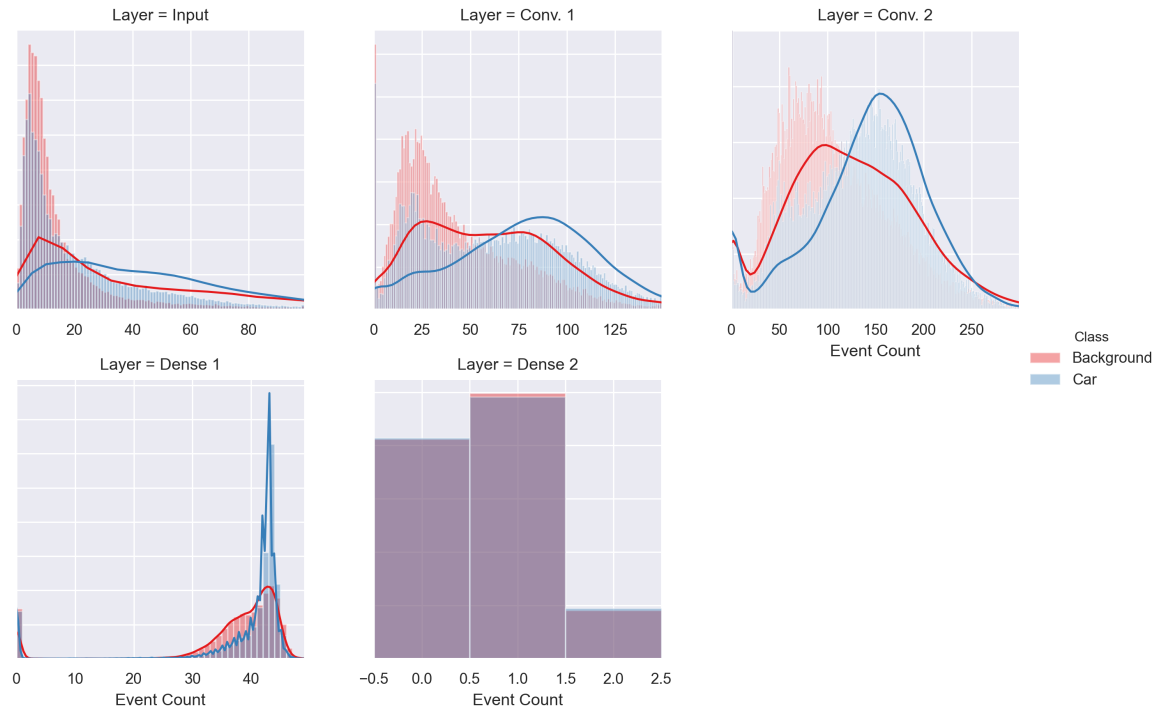


Figure 3.25: Distribution of the number of spikes per time-step and per sample class for each layer of the optimized model on the test set of N-CARS.

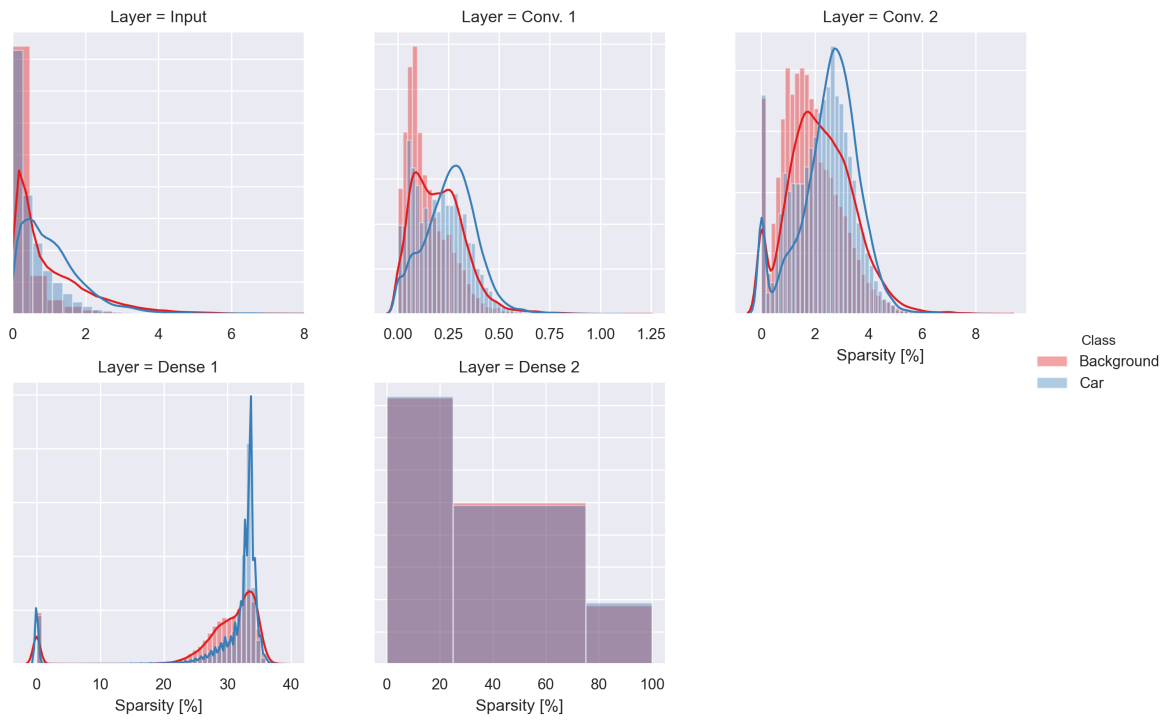


Figure 3.26: Distribution of the spike sparsity per time-step and per sample class for each layer of the optimized model on the test set of N-CARS.

As what was done in Fig. 3.10, the number of events per time-step can be drawn on a time axis. The two graphs of Fig. 3.27 represent the average number of events for each time-step and are divided depending on the layer and the class of the input sample. The difference between the average number of events in the convolutional layers between the two classes is readily visible. It further shows that the peak number of events occurs 10 ms after the beginning of the sample, which corresponds to 5 time-steps. After around 30 ms, the firing rates stabilizes for all layers. This result indicates that the network might be able to output the correct prediction before the total number of time-steps are completed ; the most relevant information in the dataset are therefore the spatial information and short-term temporal information. In an application exclusively targeted on the forward pass of the model, this would mean that the number of computations could be reduced and a correct prediction could be provided after a minimal latency of 30 ms. The prediction evaluation can then be further refined every 2 ms.

The statistics for the number of events for each layer and class are computed in Table 3.18. It further indicates the percentiles and maximum number of spikes reached in a single time-step. These information are further used in the work of Section 5.2.8 to derive the performances of the optimized real-time algorithm on this model.

The remarkable sparsity levels achieved by this network motivate the development of a program that would exclusively focus on the forward pass of the model. It would therefore leverage the low sparsity of the feature maps to perform the convolution operations with an event-based algorithm. The elaboration of a real-time inference program on an embedded platform for this model is the focus of the next chapters of this work.

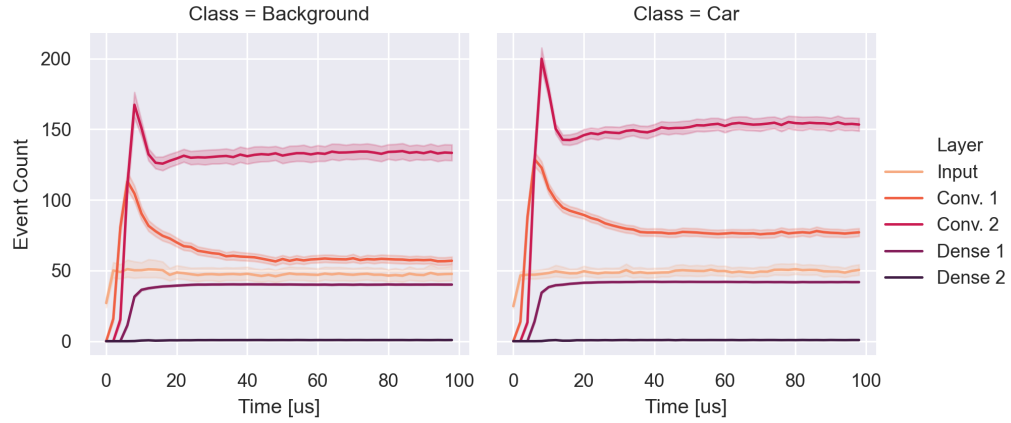


Figure 3.27: Number of events for each layer of the optimized model vs. time-step of the optimized model on the test set of N-CARS. The thick lines represent the average and the shaded area indicates the standard deviation.

Statistic	Class	Input	Conv. 1	Conv. 2	Dense 1	Dense 2
Mean	Back.	47.35	61.63	125.13	36.70	0.68
	Car	48.64	79.20	143.41	38.44	0.68
Std.	Back.	55.12	39.21	68.33	11.24	0.63
	Car	39.91	38.89	61.94	11.14	0.64
Min.	Back.	0	0	0	0	0
	Car	0	0	0	0	0
25 %	Back.	11	31	78	36	0
	Car	20	53	110	40	0
50 %	Back.	28	58	121	40	1
	Car	40	81	150	42	1
75 %	Back.	67	86	170	43	1
	Car	66	104	184	43	1
Max.	Back.	1024	365	500	49	2
	Car	594	323	453	49	2

Table 3.18: Optimized model’s statistics of the number of events per time-step, computed on the test set of N-CARS.

CHAPTER 4

ARM CORTEX-A SERIES FUNDAMENTALS

This chapter underlines the fundamental elements necessary to build a performance application on an embedded system. It is a comprehensive synthesis of elements found in ARM's Programming guides [88, 89, 90, 91], ARM's technical reference manuals [92, 93, 94, 95], the GCC compiler guide [96] and reference book of Harris & Harris [97]. The target platform is the DE10-Nano board built by Terasic [98], it is designed around the Cyclone V SoC of Altera that contains an ARM Cortex-A9 processor. The objectives of this chapter are therefore the following :

- Understand the main concepts around the architecture of an ARM processor and the memory systems around it.
- Get acquainted with extensions of the Cortex-A9 designed for high performance applications : the NEON and VFP units.
- Establish the basic framework necessary to program the Cortex-A9 and take benefit from the features of the processor.

4.1 ARM architecture and extensions

4.1.1 Introduction

ARM processors are to be found in every piece of technology : mobile phones, tablets, laptops, televisions and in other embedded systems such as cars. More than 10 billion devices integrating ARM processors had been manufactured in the world by the end of 2008, and more than 52 billion had been delivered at the end of 2013. The first ARM processor was designed in 1985 by a team of engineers led by Sophie Wilson and Steve Furber [97, 88]. The company business model evolved to a licensing model where rights to manufacture a design are sold to other semiconductor companies. The semiconductor companies incorporate those

designs with many other parts such as memory devices and other peripherals to create a complete System-on-Chip SoC.

4.1.2 Architecture versions

Compatibility is ensured between different implementations of processors via architecture specifications defined by ARM [90]. Therefore, ARM processors are compliant with a particular version of the architecture. Architecture specifications leave open important systems features such as cache sizes and timing of instructions, that are selected by the SoC manufacturer. Optional extensions can also be added to the architecture, such as the Advanced SIMD (NEON) technology that is complementary to ARMv7 and described in Section 4.3.

ARMv7 architecture also defines particular *profiles* associated with a different target market and usage.

- The *A* profile is associated with high performance processors capable of running an operating system (OS). It therefore has a virtual memory system, a NEON extension and supports the ARM and Thumb instruction sets.
- The *R* profile targets systems that requires deterministic timing and the lowest possible latency.
- The *M* profile is aimed at low cost micro-controllers with low interrupt latency, possibly only supporting the Thumb instruction set.

In this work, the primal focus is on the Cortex-A9 processor that is included in the Cyclone V SoC of the Altera DE10-Nano Board, which implements the ARMv7-A architecture.

When referring to architecture sets, the ARMv7 architecture can work with two main instruction sets : the ARM and Thumb instruction sets. The Thumb instruction set is a subset of the ARM instruction set and contains the most essential functionalities, with the advantage of encoding instructions in 16-bit words instead of 32-bit words. The advantages are a higher code density and a better cache usage of instructions, leading to increased performances. The two instruction sets can coexist in a program and critical sections may be optimized in Thumb.

4.1.3 Extensions

ARM architecture introduces a wide range of extensions aimed at offering neat solutions to specific problems [88] : Security Extensions, accelerators for Java Systems, large address spaces, etc. For the purpose of this work, 3 extensions are going to be actively used :

- The Vector Floating Point extension (VFP) includes separate registers and dedicated hardware that implements floating point arithmetic in single (32-bit) and double (64-bit) precision compliant with the ANSI/IEEE Standard [99].

- The Advanced SIMD NEON technology provides an interesting set of instructions acting on vectors of data, therefore giving the denomination *Single-Instruction Multiple-Data* (SIMD). A separate pipeline offers 8,16,32 and 64-bit integer arithmetic as well as single precision floating point arithmetic on large registers of 64 or 128-bit width. This extension therefore creates a great potential for the parallelization of critical code by offering operations, loading and storage of multiple data at once.
- The Coprocessor extension includes a dedicated interface, through 16 coprocessors, to act on peripheral controllers such as the cache, the Memory Management Unit (MMU) and Performance Monitoring Unit (PMU) for tracing.

4.1.4 Cortex A-9 processor properties

Since Cortex A-9 processor is part of the *A profile* processors, it is an appealing choice for high performance power efficient devices such as smartphones, tablets and TVs. It can be configured with up to 4 cores. The Altera DE10-Nano Board is equipped with a Cyclone V SoC that contains a Cortex-A9 Multi-Processor Unit (MPU) of 2 cores [93].

Features include :

- *Dual issue superscalar pipeline* : there are two copies of the main ARM pipeline. The datapath fetches two instructions per cycle and is able to process them in parallel if there are no dependencies between the two instructions.
- *Out-of-order dispatch* : instructions can be rescheduled in a different order than written in the program. The processor looks ahead in the code and can choose to execute an instruction based on the available pipelines, as long as dependencies are respected. This system allows to better use the pipelines and avoid dependency stalls.
- *Branch prediction* : in a pipelined processor, branches might get resolved later in the pipeline. The processor must therefore decide whether to take a branch or not before the information is available to it. In order to lower the cost of a branch mis-prediction, a *branch predictor* is used which monitors the execution flow of the program. It stores a table of the last branch destination addresses and a history on whether the branches were taken or not.
- *VFP and Advanced SIMD NEON technology extensions* : Section 4.3.
- Each core gets two 32 kB 4-way associative L1 caches for instructions and data with 64-bit data-paths : Section 4.4.
- An L2 cache controller for a 4-way associative 512 kB cache : Section 4.4.
- A main memory with 1 GB of DDR3 SDRAM.
- Debugging modules such as a PMU to record important processor events : number of instruction executed, cache usage, etc : Section 4.5.
- An MMU handling up to 4 GB of virtual memory space with configurable cache policies : Appendix B.

- A 128 entries Translation Lookaside Buffer (TLB) and two 32 entries micro TLB (μ TLB) for instruction and data : Appendix B.
- Private and Global timers for accurate time measuring and periodic task triggering.

All features are discussed in the following sections to examine potential benefits for the application.

4.2 ARM Processor and Assembly language

Assembly language is a low-level programming language where instructions have a one-to-one relationship with the machine language that the processor executes. In the workflow of a programmer, high-level programming languages such as C and C++ offer high flexibility and complex constructs to ease the elaboration of a program. This program is ultimately compiled into machine language for its execution. Assembly language comes into play when trying to optimize a code with the architecture of the processor in mind. It gives in general the highest achievable performance but at the cost of a program that is harder to understand and to maintain. In other situations, assembly language coding is necessary to perform low-level tasks such as the booting of the system or low-level peripheral operations. In this section, the fundamentals of the ARM Assembly Language are reminded in order to later seek possible optimizations of the application.

4.2.1 Processor registers

The ARM architecture contains 16 32-bit wide registers (R0-R15). Registers R0 to R12 are used for general purpose operations on data while the others have special functions :

- R13, also called the *Stack Pointer* (SP), contains the address of the top of the program stack.
- R14, also called the *Link Register* (LR), contains the return address of the routine the program is currently executing.
- R15, also called the *Program Counter* (PC), contains the location of the next instruction in the program code.

An additional register called the Current Program Status Register (CPSR) contains information about the current state of the program. It notably includes:

- The Arithmetic Logic Unit ALU flags used for conditional operations called the N,Z,C,V flags.
- The Interrupt Request (IRQ) and Fast Interrupt Request (FIQ) enable bits.
- The instruction set used (ARM/Thumb).

- The *processor mode* currently activated.

On top of general purpose registers, the ARM architecture contains specialized control registers that are accessible through coprocessors. For example, the System Control coprocessor CP15 allows to modify the important System Control Register (SCTLR) that can enable instruction and data caches, branch prediction and the MMU [90].

4.2.2 Processor modes and exceptions

The ARM architecture is *modal* which means that available registers and configurations depend on the current processor mode. The *supervisor* (SVC) mode is the default mode of the processor and is selected upon reset ; it allows modification of every configuration register. Two other important modes are the IRQ and FIQ modes that are selected when an IRQ or FIQ exception is triggered. Exceptions are signals that halt the normal flow of the program and redirect the execution towards specific handlers. The location in memory where the adequate handlers are stored is called the *exception vector table*. Exceptions may be triggered by external peripherals that require an immediate response of the processor or are generated when an error in memory occurs.

4.2.3 ARM instruction set

The ARM processor is categorized as a Reduced Instruction Set Computer (RISC) : the instruction set is reduced and aimed at general purpose operations. This often leads to a significant reduction in required internal logic compared to a Complex Instruction Set Computer (CISC) such as the famous x86 architecture [97]. Instructions include essential actions regrouped in the following list :

- *Data processing instructions* performs arithmetic or logical operations processor registers.
- *Memory instructions* allows to move data between the processor's registers and memory.
- *Branch instructions* moves the program execution to another location in the program code.
- *System instructions* modifies the features of the processor and its peripherals.

Some instructions allow to combine a bit-shift and an arithmetic or memory operation in order to speed-up increments and addressing for efficient loops and memory accesses. Finally, nearly all instructions can be conditionally executed depending on the ALU flags. The ARM instruction set is therefore simple and efficient. The following paragraphs provide more details on the different instructions available and their format.

Data processing instructions These instructions regroup the fundamental integer arithmetic and logical operations of the processor, they are performed on the 32-bit core register. The native data-type of the core is therefore the 32-bit wide integer, narrower integers require additional operations to be performed for the ALU flags. Floating-point operations are relegated to separate processing units described in Section 4.3. Apart from the multiply instructions, all data processing instructions use the ALU. The data processing instruction syntax of the ARM instruction set is described in Table 4.1 with several opcodes, braces describes intermittent elements that can appear in some instructions. The list is not exhaustive but represents the main instructions encountered in this work. All instructions can be found at [88].

`<op>{<cond>}{S} <Rd>{, <Rn>}, <Op2>`

<code><op></code>	The opcode of the data operation.	
	ADD : Addition.	
	MUL : Multiplication.	Performed on dedicated hardware and accepts only register operands.
	MOV : Data transfer.	Transfer the content of one register to another.
	AND : Logical bitwise <i>and</i> .	
	ORR : Logical bitwise <i>or</i> .	
	LSL : Left-shift.	
	LSR : Logical right-shift.	Left most bits are zeros.
	ASR : Arithmetic right-shift.	Left most bits are copies of the most significant bit (sign preservation).
	ROR : Right-rotate.	
	CMP : Compare.	Sets the ALU flags for subsequent conditional operation.
<code><cond></code>	ARM conditional operation. The instruction executes if the condition is met.	
	EQ : Equal.	
	NE : Not equal.	
	GE : Greater than or equal.	
	LE : Lesser than or equal.	
<code>S</code>	If mentioned, sets the conditional flags.	
<code><Rd></code>	Output register.	
<code><Rn></code>	First input register.	
<code><Op2></code>	Second input operand.	
	<code><Rm></code>	Simple register.
	<code>#<imm></code>	Immediate hard-coded value.
	<code><op2> <Rm></code>	Shifted register with <code><op2></code> a shift operation.
	<code><op2> #<imm></code>	Shifted immediate with <code><op2></code> a shift operation.

Table 4.1: ARM data processing instruction format.

Memory instructions As data processing instructions are not able to operate directly in memory, data elements must first be imported from memory into registers and then copied back to memory. The syntax is similar to data processing instruction as conditional operation and shifted operands are also present. The syntax for single memory transfers between memory and a register is described in Table 4.2. A different syntax for multiple transfers between registers and memory is described in Table 4.3. Multiple transfers between memory and registers typically occurs when working on the stack.

`<op>{<size>}{<cond>} <Rd>, <Addr>`

<code><op></code>	The opcode of the data operation. LDR : Copy the memory content into a register. STR : Write a register content into memory.	
<code><size></code>	Size of the memory transfer. None : Transfer a word (4 bytes). B : Transfer a single byte. H : Transfer a half-word. D : Transfer a double-word.	
<code><cond></code>	ARM conditional operation. The instruction executes if the condition is met.	
<code><Rd></code>	Data register.	
<code><Addr></code>	Address operand.	
	<code><Rn></code>	Simple register addressing.
	<code>[<Rn> <Op2>]</code> : Pre-index addressing.	<code><Rn></code> contains the base address and <code><Op2></code> contains an offset. The address for the memory access is the sum between the base and the offset. <code><Op2></code> follows the syntax of Table 4.1.
	<code>[<Rn> <Op2>]!</code> : Pre-index addressing with write-back.	The address is written back into the base register after the memory access.
	<code>[<Rn>] <Op2></code> : Post-index addressing.	The address for the memory access is in the base register. The sum between the base and the offset is written back into the base register after the memory access.

Table 4.2: ARM memory instruction format for single transfers.

`<op>{<cond>}{<mode>} <Rn>{!}, <list>`

<code><op></code>	The opcode of the data operation. LDM : Copy the memory content into multiple register. STM : Write multiple register content into memory.
-------------------------	--

<cond>	ARM conditional operation. The instruction executes if the condition is met.	
<mode>	Increment mode.	
	IA or FD : Post-increment.	The base register is incremented after the transfer.
	IB or ED : Pre-increment.	The base register is incremented before the transfer.
	DA or FA : Post-decrement.	The base register is decremented after the transfer.
	DB or EA : Pre-decrement.	The base register is decremented before the transfer.
<Rn>	Base address register.	
!	Optional write-back of the final address into <Rn>.	
<list>	List of comma separated registers. {Rd, Rm, etc}	

Table 4.3: ARM memory instruction format for multiple transfers.

Default stack implementation uses a *Full-descending stack* (FD) : the SP register points to the last filled location in memory and every new element decrements the address. For operations on the stack, the ARM instruction set therefore introduces the following syntactic sugar :

STMFD sp!, <list>	⇔	PUSH <list>	pushes registers on the stack.
LDMFD sp!, <list>	⇔	POP <list>	pops registers from the stack.

Branch instructions Branch instructions come a very simple syntax defined in Table 4.4.

B{L}{<cond>} <#imm>

L	Used to branch to a function, the return address is stored in the LR.
<cond>	ARM conditional operation. The instruction executes if the condition is met.
<#imm>	Immediate value added to PC register.

Table 4.4: ARM branch instruction format.

System instructions All reading and writing operations related with a coprocessor register are done using the special MRC and MCR instructions respectively.

4.2.4 Application Binary Interface

The *Application Binary Interface* (ABI) provides a set of rules that an assembly program must follow. They are necessary in order to produce a functional program, especially when different compilers are used. The *ARM Architecture Procedure Call Standard* (AAPCS) is one of the rules in the ABI and establishes an important convention on the usage of registers and the stack when calling a function [100]. It describes which registers are used for transferring the arguments and the return value, and which registers need to be saved by the called function (*callee*). This rule is therefore of the utmost importance when writing a function in assembly language.

- *Argument registers* (R0–R3) are used to pass function arguments. These registers are saved by the caller function and can be modified by the callee. If their capacity is not sufficient, the remaining arguments are passed on the stack. The return value of the function is passed in the R0 register.
- *Callee-saved registers* (R4–R11) must be pushed on the stack before they can be used by the callee. They must subsequently be restored at the end of the function. As it requires memory transfers, an efficient function should therefore use the argument registers first for intermediate operations.
- The SP, LR, PC registers have a distinct role explained in Section 4.2.1.

4.3 VFP and NEON Units

4.3.1 Floating-point numbers

A wide variety of programs need to perform computations involving real numbers, theoretically having an infinite precision. Since this is impossible to realize on actual machines, we have to find a way of encoding the data to limit its precision. The IEEE-754 Standard [99] is the norm for all modern computers ; it describes the behavior for the rounding of results normally requiring infinite precision. It further defines how *floating-point numbers* are stored in memory.

A floating-point number uses memory to store 3 pieces of information :

- A *sign bit* informing whether the number is positive or negative with values 0 and 1 respectively.
- An *exponent* storing the magnitude of the number.
- A *mantissa* storing the fractional part of the number.

The number is therefore expressed in the form : $x = (-1)^s \times m \times 2^e$, analogous to the scientific notation, where m and e are computed from the mantissa and exponent. The denomination floating-point therefore comes from the fact that the fractional part is not

locked in magnitude. The decimal point "floats", allowing the representation of very large and small numbers.

The standard defines 3 categories of floating-point numbers :

- The *double precision* that uses 64 bits of storage : 1 bit for the sign, 11 bits for the exponent and 52 bits for the mantissa.
- The *single precision* that uses 32 bits of storage : 1 bit for the sign, 8 bits for the exponent and 23 bits for the mantissa.
- The *half precision* that uses 16 bits of storage : 1 bit for the sign, 5 bits for the exponent and 10 bits for the mantissa.

The organization in memory for a 32-bit single precision floating-point number is shown in Fig. 4.1.

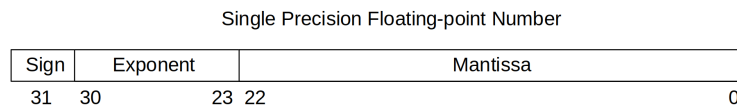


Figure 4.1: Single Precision Floating-point Format

Focusing now on the single precision format, a few more details need to be addressed regarding the exponent and mantissa format. The mantissa is interpreted as being the fractional part on the right of the binary point, with an implicit leading 1 on the left. In clear, the mantissa rather represents : $\pm 1.m$. Concerning the exponent part, it must be capable of representing both positive and negative exponents. A bias is added to accomplish this feature. In single precision, the exponent is 8-bit wide so the added bias is 127. Finally, the IEEE-754 Standard has a few special cases to represent numbers such as illegal values, positive and negative infinity, zero and subnormal numbers. Table 4.5 summarizes the single precision floating-point representation. Subnormal numbers are a smart way to create a transition from the lowest possible "normal" float (2^{-126}) to 0.0 by replacing the implicit leading one with an implicit leading 0.

Exponent (e)	Mantissa (m)	Description
0	0	0
0	$\neq 0$	Subnormal values : $(-1)^s \times 0.m \times 2^{-126}$
255	0	$(-1)^s \infty$
255	$\neq 0$	NaN values
Other	Any	Normal values : $(-1)^s \times 1.m \times 2^{e-127}$

Table 4.5: IEEE-754 single precision floating point standard, adapted from [88].

Upon data operations, the obtained mantissa might need to be rounded before fitting in the 23 bit slot of the single precision format. IEEE-754 therefore defines different rounding modes :

- *Round-to-nearest* mode finds the closest representable number to approximate the result.

- *Round-toward-0* mode is equivalent of truncating the result.
- *Round-toward- $+\infty$* consistently rounds the result towards $+\infty$.
- *Round-toward- $-\infty$* consistently rounds the result towards $-\infty$.

The ARM VFPv3 extension that is embedded in the Cortex-A9 operates on single and double precision floating point numbers. It can additionally load and store half precision floating point numbers and convert them back and forth to the single precision for computations. ARM VFP hardware further allows to disable the support of subnormal values [96], breaking the compliance with IEEE-754. As subnormal values are rarely used, this mode provides additional performance, it is called *flush-to-zero* as subnormal values are directly rounded to zero. Different rounding modes are also selectable, all these options are discussed in Section 4.6.

4.3.2 SIMD instruction principle

As seen in Section 4.2.3, data processing ARM instructions involve 32-bit operands stored in the core registers. For example, an `ADD` instruction adds the elements of two core registers or immediate values and writes the result back in a core register. However, some programs require the same operation to be repetitively carried on multiple data sources and this can produce inefficient code. The following example uses four instructions to add 8 registers (Listing 4.1).

Fortunately there exist instructions designed for such situations called *Single Instruction Multiple Data* (SIMD). It can provide a significant increase in performance because a SIMD instruction operates on data values contained in larger registers and processes them in parallel (Fig. 4.2). This organization of data is referred to as a *vector*. The same example code achieves the same operation with a single instruction operating on 128-bit registers. These instructions are therefore usually preferred in advanced data processing applications such as audio and video processing or physics simulations, where the program can be easily parallelized [89].

Listing 4.1: Scalar Add of 4 integers

```
1 add r0, r0, r4
2 add r1, r1, r5
3 add r2, r2, r6
4 add r3, r3, r7
```

Listing 4.2: SIMD parallel add of 4 integers

```
1 vadd.u32 q0, q0, q1
```

4.3.3 VFP and NEON engines

ARM NEON technology and the VFP are solutions built on the concepts of SIMD processing. The VFP unit is designed to handle floating-point data with a certain degree of parallelism while the NEON unit expands on the concept, offering higher parallelism for integer data as well as single precision floating points.

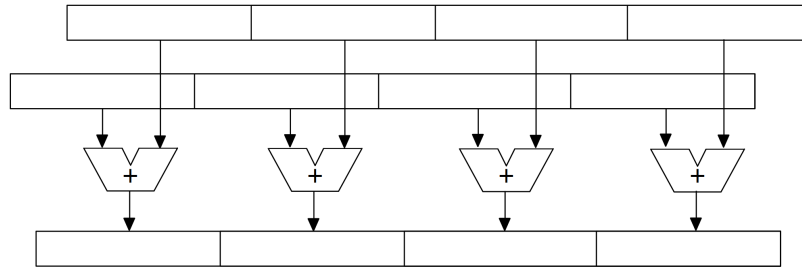


Figure 4.2: SIMD instruction principle

For ARMv7 architectures, it is not guaranteed that either a NEON or VFP unit is present. Each one of the extensions can be present independently but also be both integrated. However, if a VFP is not present, the NEON unit cannot operate on floating-point data. On the particular case of the Cortex-A9, both NEON and VFPv3 are included. It further supports half precision floating-point for both VFP and NEON units but only for load and store instructions : computations require first a conversion to single precision.

The NEON unit comprises 32 64-bit registers in a register bank that is completely separated from the main ARM execution pipeline. If a VFP unit is also present, they share the same register bank. The register bank can be reinterpreted by instructions as either 32-bit, 64-bit or 128-bit registers, which register is available depends on whether the instruction is in the NEON or VFP instruction set [89]. The overlap between register denominations is shown in Fig. 4.3. There are 16 *quad-word registers* numbered as Q0-Q15, 32 *double-word registers* numbered as D0-D31 and only 32 *single-word registers* numbered as S0-S31 that corresponds to the first 16 double-word registers.

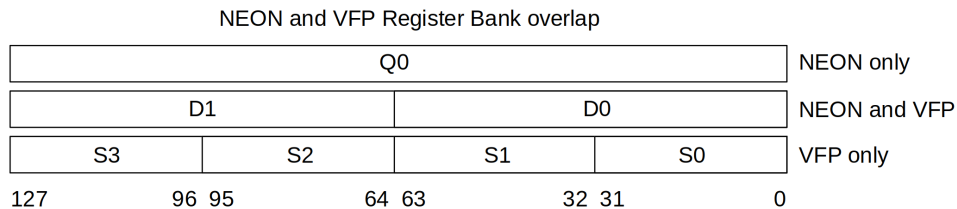


Figure 4.3: NEON / VFP register bank reinterpretation

A register always comprises data *elements* of the same data-type into a *vector of elements*. The location that a particular element has in the vector is called a *lane*. Available data-types are reported in Table 4.6. Thus, a double-word register can contain either :

- 8×8 -bit elements.
- 4×16 -bit elements.
- 2×32 -bit elements.
- 1×64 -bit element.

Obviously, a quad-word register has double the amount of elements of a double-word register.

NEON Data-types				
	8-bit	16-bit	32-bit	64-bit
Unsigned integer	U8	U16	U32	U64
Signed integer	S8	S16	S32	S64
Integer of unspecified type	I8	I16	I32	I64
Floating-point number	N.A.	F16	F32	N.A
VFP Data-types				
Floating-point number	N.A.	F16	F32	F64

Table 4.6: NEON and VFP data-types.

SIMD instructions therefore applies the same operation independently on each lane in parallel. Because of the limited size of registers, a more compact data-type allows a higher parallelism.

To summarize, the main differences between the VFP and NEON units to bear in mind are :

- The NEON unit only processes multiple data at the same time in a pure SIMD fashion in D and Q registers.
- The VFP unit processes scalar and vector data in S and D registers.
- The NEON unit supports computations on integer and single precision floating-point data. It does not fully implement the IEEE-754 standard since it operates in flush-to-zero mode. Its rounding mode is fixed to round-to-nearest.
- The VFP unit supports computations on single and double precision floating-point data. It is IEEE-754 compliant.
- The VFP unit performs complex operations such as a floating-point divide and square-root whereas the NEON unit cannot.

The NEON unit has its own execution pipelines that executes different kinds of instructions. Despite having its own pipelines, the NEON unit is still part of the processor core and uses ARM registers for memory addressing in SIMD load and store instructions. The NEON unit contains :

- An integer execute pipeline.
- A single precision floating-point pipeline.
- A load/store and permute pipeline.

The NEON instructions first go through the ARM pipeline and then rest in the NEON instruction queue before entering one of the NEON pipelines [89]. As long as the queue is not saturated, the processor is able to execute both ARM and NEON instructions ; when saturation happens it stalls the entire pipeline. Knowing the details of this architecture, the NEON unit is therefore capable of *dual-issuing* load/store/permute instructions and data processing instructions. These two kinds of instructions should preferably be interleaved in the code for optimal use of the pipelines. Furthermore, different instructions do not require

the same amount of cycles to complete : some instructions such as `VADD` may only need 1 cycle to complete (if no dependency stall occurs) while others such as `VSQRT` might require as much as 28 cycles [95]. An optimal scheduling of instructions is therefore paramount to obtain an efficient program. In some cases, there could be substantial additional latencies if instructions require results generated by previous instructions. For example, the floating-point multiply and accumulate instruction (`VMLA.F32`) needs 10 cycles for the result to be accessible by other instructions while it only occupies 2 cycles in the pipeline.

4.3.4 NEON instruction syntax

The NEON instructions can be divided in two main categories : *data processing* and *data movement* operations. A comprehensive list of instructions is available in *ARM's NEON Programming Guide* [89]. Operations include :

- Data processing
 - Addition and subtraction.
 - Multiplication.
 - Multiply and accumulate : it combines a multiplication between two vectors with the addition or subtraction of a third vector.
 - Shift left and shift right.
 - Logical operations.
 - Compare operations.
 - Data transfer between registers.
- Data movement
 - Load and store between registers in memory, possibly with interleaving.

Unlike the main ARM pipeline, compare operations cannot modify the `CPSR` register. Instead, the NEON pipeline has its own status flags stored in the *Floating-point status and control register* (`FPSCR`) that are modified by scalar VFP instructions. They can be transferred to the ARM pipeline via the `VMRS` instruction for further conditional execution. When dealing with vector data, NEON instructions cannot modify the `N,Z,C,V` flags of the (`FPSCR`) ; instead it produces an output bit mask that can be used for bit transfers between registers. The instruction format of data processing operations is synthesized in Table 4.7.

V{<mod>}<op>{<shape>}{<cond>}{.<dt>} <dest1>{, <dest2>}, <src1>{, <src2>}

<mod>	It causes a modification of the result at the end of the operation. It is only applicable on integer data-types.	
	Q : Saturation	The output cannot overflow and saturates at the maximum or minimum value of the associated data-type.
	H : Halving	The output is divided by two.
	D : Doubling	The output is multiplied by two.
	R : Rounding	The output is rounded before a truncation operation (either halving or narrowing).
	None : Normal	The output is kept the same.
<op>	The opcode of the data operation.	
	ADD : Addition.	
	MUL : Multiplication.	
	MLA : Multiply and accumulate.	Equivalent to a multiplication followed by and addition, this instructions is optimal for performing weighted sums.
	MOV : Data transfer.	Transfer the content of one register to another.
	etc	All instructions can be found at [89].
<shape>	It dictates the register width of the input and output operands. It is only applicable on integer data-types.	
	L : Long	Input width : double-word Output width : quad-word.
	W : Wide	Input width : double-word and quad-word Output width : quad-word.
	N : Narrow	Input width : quad-word Output width : double-word.
	None : Normal	Input and output registers have the same width.
<cond>	ARM conditional operation.	
<dt>	The data-type of elements in input and output registers (Tab. 4.6).	
<src1>, <src2>	Input register operands.	
	Dd : Double-word register	Operation on D register number d.
	Qd : Quad-word register	Operation on Q register number d.
	Dd[x] : One lane of a double-word register	Selects lane x of D register number d and duplicates it for operation (see Fig. 4.4). It is only available as <src2>.
<dest1>, <dest2>	Output register operands.	
	Dd : Double-word register	Operation on D register number d.
	Qd : Quad-word register	Operation on Q register number d.

Table 4.7: NEON processing instruction format.

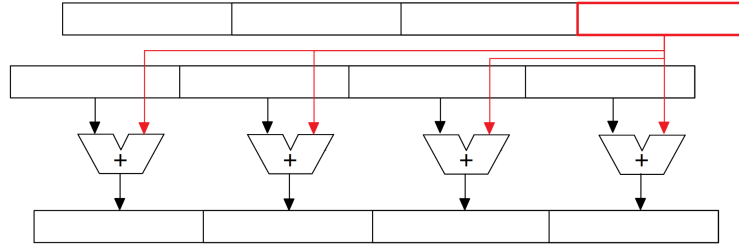


Figure 4.4: SIMD instruction with single lane operand.

The instruction format of data movement operations is slightly different : it ignores the data-type inside the registers and only cares about the lanes' width. It takes lists of registers and a memory address as operands and the format has a new placeholder characterizing the interleaving of elements in memory. The format of data movement instructions is shown in Table 4.8. As it is difficult to visualize, a de-interleaving example is shown in Fig. 4.5 for a 16-bit data structure of size 2.

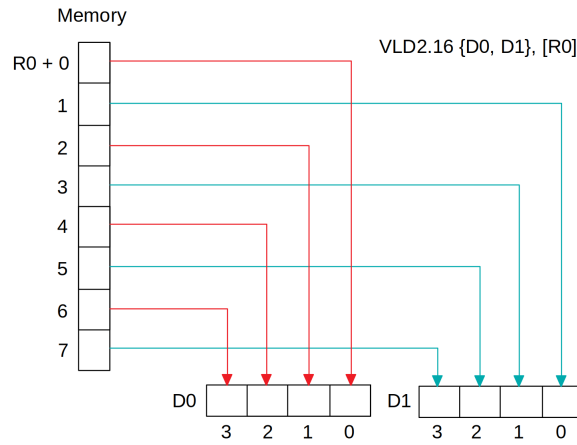


Figure 4.5: De-interleave behavior upon memory load.

V<op>{<n>}{<cond>}{.<size>} <list>, [<Rn>{@<align>}]

<op>	The opcode of the data operation. LD : Data load ST : Data store
<n>	n, called the <i>structure size</i> , is one of 1,2,3 or 4 is the size of the data structure in memory. The structure elements are interleaved when written into memory and de-interleaved when read from memory.
<cond>	ARM conditional operation.
<size>	The data-type size of elements in the list of registers (size of types in : Tab. 4.6).
<list>	List of double-word registers whose length is at most equal to 4 and depends on the data-type and structure sizes [89]. {Dd, D(d+1), etc} The data structure elements in memory maps to the lanes of one or more D registers. {Dd[x], D(d+1)[x], etc} The data structure elements in memory maps to lane number x in one or more D registers. {Dd[], D(d+1)[], etc} Only available for the LD operation, the data structure elements in memory maps to one or more D registers and the elements are duplicated on all lanes of the register.
<Rn>	The ARM register that contains the target memory base address
<align>	Optional memory alignment hint for fast memory access

Table 4.8: NEON processing instruction format.

4.3.5 NEON optimization

In practice, programmers would rarely write a piece of code directly in NEON Assembly language : it is rather difficult to master due to scheduling issues and register allocation and it is hard to maintain and modify afterwards. For that reason, programmers often write code in higher level programming languages in order to focus on the algorithmic aspect. After that, the task of the compiler is to translate the language into Assembly operations, assign registers and stack usage and optimally schedule the instructions for the given processor. Apart from programming directly in Assembly language, there are therefore two ways a programmer can find to benefit from the specialized NEON hardware and related instructions described above.

1. The compiler can be parametrized to produce SIMD assembly code from C/C++ language.
2. A specialized NEON library exists and offers NEON *intrinsics* functions.

Vectorizing compiler Provided that the compiler is correctly parametrized for a given ARM processor (see Section 4.6), it has the ability to *vectorize* C/C++ code to produce NEON instructions. However, this ability heavily relies on good programming habits and coding styles that renders the code easily vectorizable. This mainly includes giving the compiler additional information about the intents of the programmer for particular variables, via builtin compiler functions and keywords, so that the compiler is able to safely parallelize code. As referred in [88] and [89], an efficient coding style for implying SIMD instructions include :

- Having the number of loop iterations known by the compiler (no **while** loops or **break** statements).
- Having the number of loop iterations a power of 2.
- Limited number of statements inside a loop.
- No function calls inside a loop.
- No conditional statements inside a loop.
- Avoid multi-dimensional array operations.

All techniques are utilized in Section 5.2.3.1 when optimizing the program of our application with the vectorizing compiler.

NEON intrinsics The NEON intrinsics are C/C++ functions that have a syntax very similar to the NEON instructions. It also introduces new data-types that maps to NEON D and Q registers. It provides a way to directly generate NEON instructions while keeping the advantages of a high level programming language :

- The compiler performs register ARM and NEON register allocations.

- The compiler reschedules the instructions generated by intrinsics to minimize pipeline stalls.
- The intrinsic can map to different Assembly instructions depending on the situation.

The last point can sometimes be regarded as a disadvantage since intrinsics can still produce a different code than the one intended by the programmer. At this point, the last resort is to program directly in Assembly language for a total control over the produced code. A comprehensive list of all available intrinsics and data-types is available in [89].

The intrinsics are available when including the `arm_neon.h` C header in the the code, it provides several two new sorts of data-types : the *vector* and *array* data-types. Table 4.9 detail the nomenclature of the vector and array data-types.

`<type><size>x<number_of_lanes>{x<length_of_array>}_t`

`<type>` The data-type of every element in the vector.
 `int` : Integer
 `uint` : Unsigned integer
 `float` : Floating-point

`<size>` The number of bits of an element. The allowed combinations with `<type>` are listed in Table 4.6.

`<number_of_lanes>` The number of elements in the vector to fill a D register or a Q register.

`<length_of_array>` If present, the data-type therefore represents `struct` containing an array of consecutive D or Q registers. This is illustrated in Fig. 4.6 for a `float32x4_t` vector.

Table 4.9: NEON intrinsics vector and array data-type format.

```
1 struct float32x4x2_t
2 {
3     float32x4_t val[2];
4 };
```

Figure 4.6: NEON intrinsics array structure for the `float32x4_t` vector data-type.

The vector data-types are used for input and outputs of NEON intrinsics functions. These functions translates to NEON instructions and therefore have a similar format, detailed in Table 4.10.

`<op><flags>{q}{_lane}_<type>_t`

<code><op></code>	The opcode of the associated NEON instruction, as detailed in Tables 4.7 and 4.8.
<code><flags></code>	It regroups <code><mod></code> and <code><shape></code> modifiers of Table 4.7 or the <code><n></code> modifier of Table 4.8.
<code>q</code>	If present, it indicates that the function uses vectors mapping on Q registers. Otherwise, D registers are used.
<code>lane</code>	If present, indicates that the operation involves only single lane in D registers.
<code><type></code>	The data-type of every element in the input and output vectors. The allowed combinations are listed in Table 4.6.

Table 4.10: NEON intrinsics function format.

The array data-type intervenes only in load and store functions that uses an interleaving modifier (`<n>`). As presented in Fig. 4.5, the values in memory are mapped with the array of D or Q register in the array data-type.

This concludes the explanations on VFP and NEON units. The next section takes a radical turn and focuses on the memory systems connected with the ARM processors.

4.4 Memory systems

4.4.1 Motivations

When processors were first developed, the access time of memories and the processing speed were of the same order of magnitude. Processors today operate at frequencies much higher than before but the frequency of external memories and busses has not increased by the same factor (Fig. 4.7).

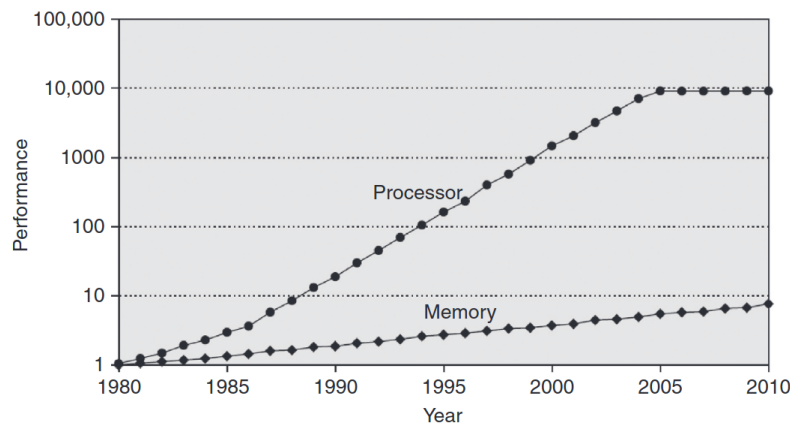


Figure 4.7: Diverging processor and memory performances, reproduced from [97].

As detailed in [97], computer memories are built from dynamic RAM (DRAM) or static RAM (SRAM) and ideally, a computer memory system must be large, fast and cheap to manufacture. In practice however, memory systems have only two of the three attributes at a time : it is either fast and small, slow and large or has both attributes at a higher cost. An efficient technique to get the best out of both worlds is to combine cheap memories with different properties. The *Main Memory* is constituted generally of DRAM chips and has a storage capacity of several GBs for an access time in the order of tens or even hundreds of processor cycles. On the other hand, a smaller and faster memory called the *cache* stores instructions and data that are most used by the processor. It is built out of SRAM and placed on the same chip as the processor. It has an access speed comparable with processors at the cost of a storage capacity of only a few tens of kBs. Because the cache can only hold a small subset of the main memory, it needs to store the data of interest along with its address in main memory. When the program running in the processor needs to access a particular address in memory for a read or write operation, the processor will first look into the cache to see if the information is available in it. This allows a significant reduction in access time as well in power consumption since external signals are not driven to access the main memory.

Different architectures of caches exist in order to benefit from *temporal locality* and *spatial locality* that the data may have in the program [97], [88].

- The temporal locality property means that programs tend to re-use the same data over time. For example a program might want to read an address from the memory, process the data and then write the data back on the same address quickly after. Also, if the code contains loops it means that the same instructions are going to be executed again in a short time.

- The spatial locality property means that programs tend to access at the same time data that are close together in memory. This is for example the case when processing elements out of an array of values or when reading instructions that follow each other in memory.

4.4.2 Cache architecture

As mentioned before, the cache is required to hold both data and associated addresses as well as some status information. Therefore, for each value in the cache, the corresponding address is stored next to it and is denominated as the *tag*. When reporting the cache total capacity (e.g. 32 or 512 kB), the memory required to store the tag is omitted.

It would be highly inefficient to store one word of data (32 bits) with one address (32 bits), instead a contiguous block of data can be stored for a corresponding tag. This is known as a *cache line* and this arrangement allows to benefit from the spatial property that data may have : when working on an array, blocks of data are therefore available at the same time. Therefore, only the top bits common to each word in the line are used for indexing in the cache and storing the tag. The bottom bits, named the *offset*, are used for indexing within the cache line. Status bits are attached to the cache line to provide additional information. The *valid bit* informs whether the data held is usable. At initialization for example, the cache is empty and each line is marked as invalid. The *dirty bit* informs whether the data held in the line is different (newer) from the one in main memory.

From that point, there exists several ways of implementing a cache [88]. A *direct mapped cache* maps each location in the main memory to a single line in the cache. Since caches are significantly smaller than main memories, several addresses in memory point to the same cache line (Fig. 4.8).

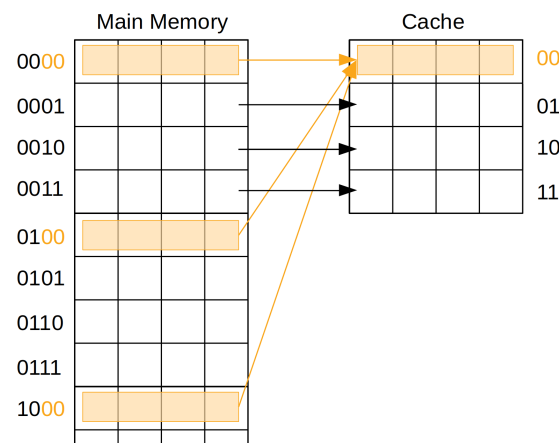


Figure 4.8: Direct mapped cache, adapted from [88].

The mapping is determined by the bottom bits on the left of the offset ; the number of cache lines is dependent on the total cache size. The top bits that are left over are used as a tag. For example, let's consider a direct mapped cache with a capacity of 32 kilobytes and a line size of 32 bytes. Since words are 32 bits (4 bytes) long, the `bit[1:0]` are used for accessing a byte in the word. With 8 words in a cache line `bits[4:2]` are used as the offset. The number of lines is obtained by dividing the cache capacity with the size of a cache line,

which leaves 1024 lines so `bits[15:5]` are used for indexing a cache line (*index*). This yields `bits[31:16]` as a tag, which is illustrated in Fig. 4.9.

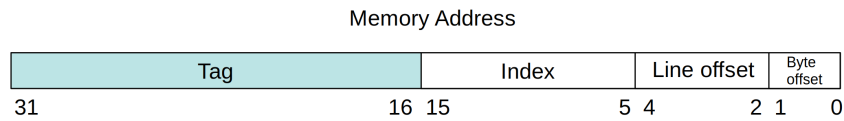


Figure 4.9: Example of cache address indexing.

The cache reading / writing process, called the *cache lookup* is therefore the following : when accessing an address in memory, dedicated logic extracts the index to access the corresponding cache line. It then compares the tag and one of two things can happen :

- The line is valid and the tag corresponds with the one stored in memory, therefore the cache *hits*. The requested word is extracted using the line offset and is transferred to the processor pipeline.
- The line is invalid or the tag does not correspond, therefore the cache *misses*. If it was invalid, it is directly replaced with data from the main memory, this is called a *cache linefill*. If the cache had already valid data, it must be first written back to main memory and this is called a *cache eviction*. This process is explained below when talking about cache policies.

However, a direct mapped cache is rarely used due to the following problem. If the program requests several pieces of data that happen to share the same cache line, only one of them is going to remain in the cache. This can lead to excessive accesses to the main memory, cancelling all the benefits that a cache system brings. This phenomenon is called *cache trashing*. For example, the following small program can suffer from trashing if any of the top addresses of arrays `a`, `b` or `c` share the same cache line. At each iteration of the loop, the same cache line going to be overwritten alternatively by `a[n]`, `b[n]` and `c[n]`, ruining the advantage of data re-use in the cache line.

```

1 void add(int *a, int *b, int *c, const unsigned int len)
2 {
3     for (unsigned int n{0}; n < len; ++n)
4     {
5         c[n] = a[n] + b[n];
6     }
7 }

```

To counteract cache trashing, a different cache organization can be used : a *set-associative cache* [88]. The cache is now divided into a number of identical regions called *ways* and main memory locations map to ways rather than a specific cache line, as illustrated in Fig. 4.10. The ways are indexed in the same manner and each one holds a different tag. In this configuration, for a fixed cache size, there is now a smaller number of cache lines but several data with the same cache index can now coexist inside the cache, reducing the probability of trashing. The ensemble of cache lines that share the same index is called a *set*.

A cache lookup now involves comparing the requested tag with several all tags of different ways to check if the data is available. If the cache misses, it can fill the missing data into an

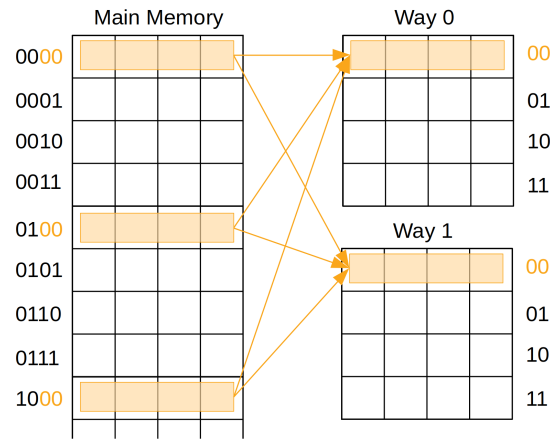


Figure 4.10: Set-associative cache, adapted from [88]

available way. Building on the previous example, the Cortex-A9 instruction and data caches are said to be 32 kB 4-way associative caches with a 32 bytes line size. It therefore contains 256 cache lines but can now contain up to 4 pieces of data with the same cache index. This organization allows to better take benefit of the temporal locality as data are susceptible to remain in the cache for a longer time.

A complete illustration of a 2-way associative cache with a line length of 4 words is displayed in Fig. 4.11. It is annotated with the vocabulary introduced in this section.

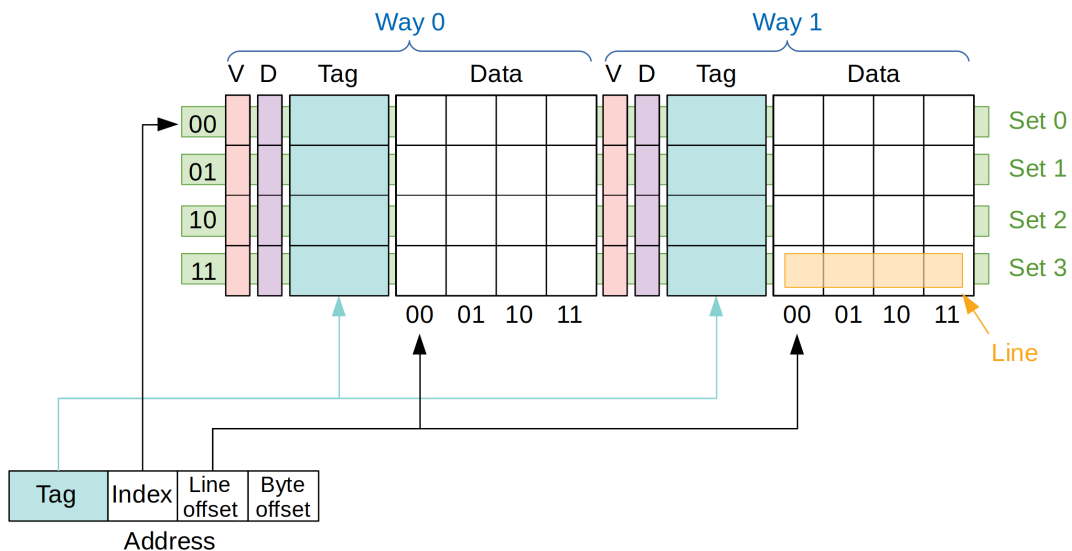


Figure 4.11: 2-way associative cache illustration, V and D represents the valid and dirty bits. Byte indexing within a word of a cache line is not represented.

Finally, a third type of cache is the *fully-associative cache* [97]. It is a set-associative with a single set only. There is therefore a single cache index and as many ways as there are cache entries. The fully-associative cache therefore takes fully advantage of the temporal locality that the data may have and ignores completely its spatial locality. On the other hand, they require a larger circuitry to perform the tag comparisons. Fully-associative caches can be used for implementing *table lookaside buffers* that needs a high level of temporal locality (Appendix B.3).

4.4.3 Cache policies

The cache control registers can be modified to dictate the cache's behavior when encountering several situations.

- The *cache allocation policy* dictates the action of the cache when there is a cache miss.
 - A *read allocate policy* allocates a cache line only in the event of a read miss. If a write misses, it is directly written into the main memory.
 - A *write allocate policy* allocates a cache line when there is either a read or write miss.
- The *replacement policy* dictates the selection process for the way that needs to be replaced by incoming data when all ways are already valid and dirty in a cache line. The replaced way is named the *victim*.
 - A *cyclic replacement policy* selects a different way each time this event occurs, with the help of counter.
 - A *pseudo-random replacement policy* selects a victim in a random fashion.
 - A *least recently used policy* tracks the usage of each way and selects the one that was least recently used.

The Cortex-A9 does not allow modification of the replacement policy and uses the pseudo-random policy by default [93]. This policy is often preferred as it generally provides the same performance across a wide range of different programs.

- The *write policy* further specifies the behavior of the cache when writing data.
 - A *write-through policy* indicates that cache and the main memory must remain identical (*coherent*). Writes are therefore performed on both the cache and the main memory.
 - A *write-back policy* indicates that the cache and the main memory are allowed to be non-coherent. If data is written, it is modified directly in the cache and marked as dirty and only written back into main memory when evicted.

The Cortex A-9 processor allows to impose different cache policies for different memory regions thanks to the MMU and its use of *virtual addresses* [88]. The notions of virtual memory and TLBs are complementary to the concepts of this section but are relayed in Appendix B.

4.4.4 Cache buffers

In order to further increase the gain in performance offered by a cache system, read and write operations to the main memory can be relegated to buffers. A buffer is a hardware block inside the core that collects addresses and data that needs to be read or written into main memory. With this system, the processor does not need to wait for read or store operation to complete and can carry on with the execution of the next instruction. Provided that

number of load and store instructions are sufficiently low and spaced, the buffer will procure a strong impact. On the other hand, if the buffer is filled it will inevitably cause a pipeline stall. Write buffers often support merging of multiple data writes to main memory into a single burst, reducing traffic to main memory [88].

The Cortex-A9 data cache contains two 32-byte linefill buffers to manage misses and another 32-byte eviction buffer. For write operations, it has a 4-entry 64-bit merging store buffer. Indeed, the data-path between the L1 cache and the lower level memory is 64-bit wide. Additionally, the Cortex-A9 features a particular kind of buffer called the *pre-fetch buffer* whose purpose is to load instructions and data ahead of time for increased performance. Its operation is mostly transparent but allows for interaction with the assembly instruction PLD. It performs a cache lookup and might produce a linefill in advance, while the processor is still executing other instructions.

4.4.5 Cache hierarchy

A modern processor typically includes several caches in order to moderate the impact of a cache miss. In ARM processors, a *level-1* (L1) cache is connected on the same chip as the processor. They have an *Harvard architecture* which means that it handles separately and concurrently instructions and data, with separate buffers. The L1 caches have sizes that vary in the tens of kilobytes, allowing a memory access in single cycle [94]. In the Cyclone V SoC, the 2 Cortex-A9 cores have separate L1 caches.

Additionally, processors often have a *level-2* (L2) cache that fits in between the L1 caches and the main memory. ARM systems provides L2 caches with storage capacity in the order of a few hundreds of kilobytes and that have a Von Neumann architecture where instructions and data are not separated. This cache is slower than the L1 but still orders of magnitude faster than the main memory, the Cortex-A9 offers a L2 that has latency of 6 cycles in the best case [93]. A simplified schematic for this memory system is shown in Fig. 4.12.

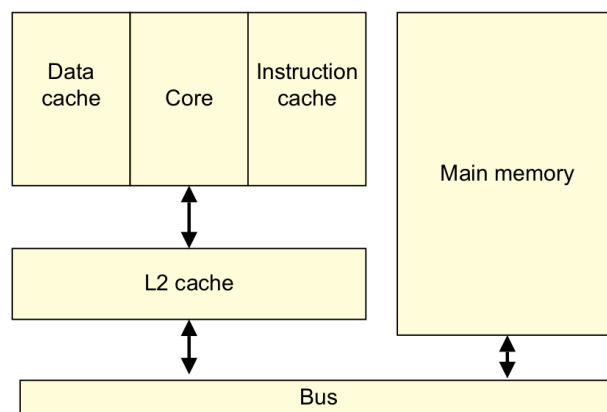


Figure 4.12: Two level memory system with a L1 Harvard cache, reproduced from [88].

In a multilevel cache system, the MMU controls cache allocation and write policies between L1 and L2 caches (*inner cache policies*) and between L2 and main memory (*outer cache policies*). It further allows an exclusive cache policy where a piece of data cannot live in the two caches at the same time, increasing the total available cache size [92]. The replacement

cache policy is modified so that a L1 eviction always end up in the L2 cache even if the data is marked clean. Upon reading a dirty line in the L2 cache, it simultaneously triggers a linefill of the L1 cache and a write-back to memory.

Finally, the L2 cache of the Cortex-A9 is equipped with a preload engine (PLE) [94], capable of transferring large blocks of memory directly from main memory. Its operation is similar to the one of the pre-fetch buffer of the L1 cache but differs with the fact that it can load in advance multiple cache lines with a single request.

4.4.6 Cache drawbacks

With all the features described above, a cache system instantly appears as a large benefit that can enhance any application. There are however a few caveats that one should bear in mind, the most disturbing of which is that the program execution becomes non-deterministic. This is due to the fact that caches have inherently less memory than the main memory and the movements of data are dictated by the pseudo-random cache replacement policy. Therefore, a particular piece of code can end up having an execution speed that varies greatly depending on the current cache usage. Additionally, depending on the scheduling of instructions, operations might have to wait for the buffers to complete their reading or writing operations. It is therefore very difficult to accurately measure timing characteristics for a program. However, smart pre-fetching of data before their execution can reduce this impact by assuring that data is to be found in the cache most of the time.

Finally, another miscellaneous behavior appears when dealing with multi-core systems. If a first core processes a block of data that is then kept inside his L1 cache, a second core accessing the same addresses will fetch expired data (*stale* data). This problem of coherence is embarrassing and can be solve either with a careful planning of core actions in the program or by using write-through policies.

4.4.7 Performance measurements

To conclude this section on memory systems, important metrics and recommendations are introduced and are used in the following sections.

First of all, the most generic performance metrics that can be used to measure the effectiveness of the cache system is the cache *hit rate* or *miss rate* [97]. The cache hit rate, expressed as a percentage, is the ratio between the number of memory accesses that hits in the cache and the total number of memory accesses. Equivalently, the cache miss rate computes this ratio but with respect to cache misses. As the two metrics can be computed from one another, we are going to discuss solely in terms of miss rate.

$$\text{Hit Rate} = \frac{\text{Number of cache hits}}{\text{Number of memory accesses}} \quad \text{Miss Rate} = \frac{\text{Number of cache misses}}{\text{Number of memory accesses}}$$

A low cache miss rate is synonym of good data re-use in the cache and leads to high performances. In fact, there is a large gap that separates the cache hit that returns the data within a few cycles, from the cache miss for which often tens of cycles are necessary. Improving

algorithms to have fewer cache misses is thus one of the most important optimizations [88]. A program can suffer a cache miss for different reasons [97] :

- A *compulsory* miss happens when a piece of data is accessed for the first time. It therefore triggers a linefill and hopefully the subsequent elements of the fetched line are used later in the program.
- A *capacity* miss is too small to hold all concurrently used data at the same time.
- A *conflict* miss is caused when a piece of data is evicted by another while it was still needed. This was also referred as cache trashing.

As the performance of an algorithm with respect to the cache is linked with ideal data usage, the biggest gains in performance are achieved by implementing the following recommendations [88], [89], [96].

- Operations involving data structures such as C/C++ **struct** and arrays can be optimized by having their base address at the beginning of a cache line. Therefore, a single linefill provides the data for the entire **struct** or a block of the total array.
- Whenever possible, access in a data structure should be sequential to benefit from the linefills.
- In operations involving multiple data structures, data that are used at the same time should be kept close together in memory to avoid cache trashing. One effective way to achieve this is to interleave in memory the data that are used concurrently.
- Pre-fetch hints through the PLD instruction in large deterministic loops can provide a large benefit as they can remove compulsory misses.
- Optimal usage of the instruction cache is achieved by re-using the same piece of code to perform similar operations in the program. This is done by creating functions to avoid duplicating code.
- Smaller code can potentially cache better than larger codes. Using the 16-bit Thumb instruction set might therefore provide benefits in speed. However there is a trade-off to be acknowledged since the Thumb instruction usually needs more instructions to perform equivalent tasks.

Those recommendations along with the ones of Section 4.3 are exploited in further details in Section 5.2.

4.5 Performance Monitoring Unit

This section introduces the solution ARM's processors offer to realize software *profiling*. In general terms, profiling is an ensemble of techniques that allows identification of critical sections of code in terms of memory usage, resource's efficiency and execution time. It therefore helps identify *bottlenecks* that need further optimization.

The Cortex-A9 processor, along with other Cortex-A and Cortex-R processors, incorporates hardware designed for that extent called the *Performance Monitoring Unit* (PMU) [90]. Its role is to offer multiple tuneable counters to record core events such as the number of cycles, the number of instructions executed, the number of memory accesses, the number of cache hits and many others. Performance metrics such as instructions per cycle and miss rates can then be computed from the sampled events.

These features operates independently of the normal program flow and do not interfere with it, providing accurate results. The programmer can parametrize up to 6 different counters through the CP15 coprocessor to monitor specific events and enable, disable or reset the counters during the program's execution.

Additionally, the unified L2 cache supplies independent event monitoring regarding its own memory accesses [92]. L2 event monitoring is enabled through configuration of the related bits in the *Auxiliary Control Register*.

In order to display and interpret the result, the *ARM Development Studio* (DS-5) includes a dedicated software for that matter called *Streamline* [91]. It provides graphical representations to assess counters evolution through the program's execution time and allows timed annotations to mark the time duration of different pieces of code. The interface is provided through a small library called *Barman*. The PMU counters and other custom counters can then be sampled by a subroutine call and their statistics recorded in an allocated buffer in memory. This system therefore introduces a slight perturbation in the counter values due to the sampling code. A complete list of available events is available in [90] for the MPU and [92] for the L2 cache.

4.6 Bare-metal framework

This section closes the chapter on the fundamentals required to build an efficient embedded application and focuses on the practical aspects of getting a functional *Bare-metal* C/C++ application on the Cortex-A9, comprising all elements described in the above sections. This step provides a stable framework that allows to develop a high performance Spiking Neural Network in the following chapter.

This section begins by listing the important features of the target platform : the DE10-Nano Board of Altera. Then the different elements necessary for the boot sequence are described in order to get an initialized hardware. In order for the target application to benefit from the ARM extensions and to fit in the board memory, the required compiling and linking options are presented. Finally, when the target application begins its execution, remaining initializations and configurations are described.

The application is chosen to operate in a bare-metal fashion, without the need of an OS. Even though there exists Real-Time OS (RTOS) supported on the DE10-Nano Board, a bare-metal application provides finer control over the memory allocation during execution. Furthermore, the potential of an RTOS really comes into play when the target application requires *pre-emption* of a set of tasks over others. This paradigm is not considered in this work since it rather focuses on the building blocks of an embedded machine learning application : fundamental operations are optimized on a single core. Working on a single

core also allows to avoid coherence issues when several cores work on the same piece of data.

For reasons that become more apparent in the next chapter, the application is programmed in C and C++ languages. Interesting libraries such as the NEON intrinsics, Altera maintenance functions and I/O FAT functions are available in C. The C++ language builds upon the C concepts and further introduces other paradigms such as *object oriented programming* and *template meta-programming*. The two languages can be mixed in a single program and compiled using the widely used GCC compiler.

4.6.1 DE10-Nano Board features

The DE10-Nano Board of Terasic [93] is a hardware platform built around the Cyclone V SoC built by Altera. The SoC combines a *dual-core* ARM Cortex-A9 and a field programmable gate array (FPGA). The processor is also referred as the *hard processor system* (HPS) in this hybrid system. The board has the following characteristics :

- Hard Processor System
 - 800 MHz Dual-core ARM Cortex-A9 processor.
 - NEON and VFPv3 units.
 - 32 kB of L1 Instruction cache and 32 kB of L1 Data cache.
 - 512 kB of L2 unified cache.
 - 64 kB of On-Chip RAM (OCRAM).
 - 64 kB of On-Chip Boot ROM.
 - 1 GB of DDR3 SDRAM connected with a 32-bit data bus.
 - Secure Digital/Multimedia card (SD/MMC) slot.
 - UART to USB connector.
- FPGA
 - 110 thousand logic elements.
 - 112 variable precision DSPs.
 - 145 configurable I/O pins.
 - 224 multipliers.
 - 32-bit Fast Fourier Transform engine.

The FPGA potential is not considered further as the NEON unit is rather preferred for its integration in the ARM pipeline. The memory map of the Cyclone V processor is examined in Fig. 4.13.

The Cyclone V has a memory space of 4 GB that maps different devices. Peripherals of the FPGA and the HPS are mapped in the upper part of the memory space. It comprises all peripheral's control registers. Below is the SDRAM and boot regions. If a *flat virtual space* is considered, only the region below address 0x4000 0000 is of interest for memory storage since the DE10-Nano board only contains 1 GB of SDRAM. The boot region can be mapped to two separate memories :

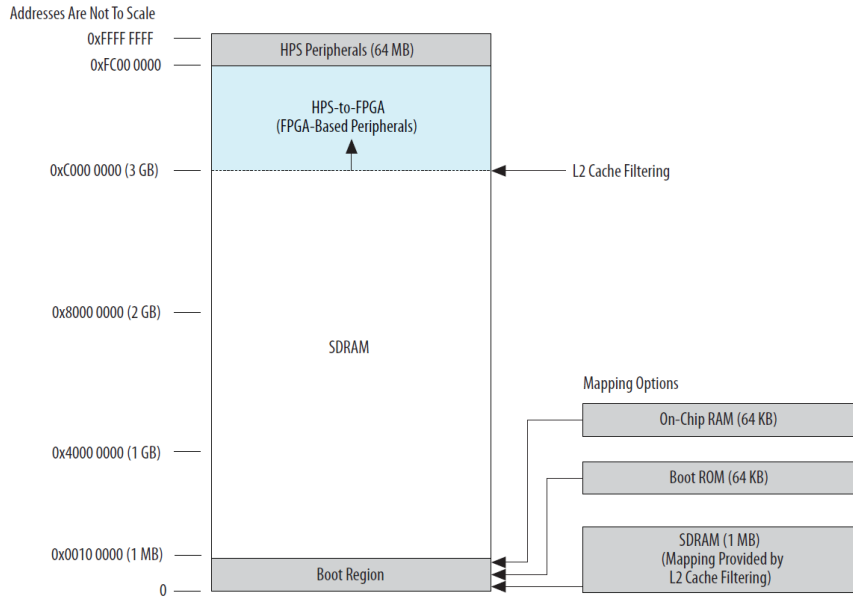


Figure 4.13: Memory map of the Cyclone V MPU, reproduced from [93].

- The *On-Chip RAM* is a small memory integrated in the core that is 64 kB in size.
- The *Boot ROM* is 64 kB in size and contains the code needed by the HPS upon restart.

The mapping is configurable through the L2 control registers, this remapping is mainly useful during the boot sequence.

4.6.2 Boot flow

Between the cold state of the Cortex-A9 processor and the moment our optimized application starts to execute, a number of intermediate steps are necessary to initialize and parametrize the hardware. How is the program loaded in memory ? How are the caches and NEON unit enabled ? This section aims at explaining the sequence of steps, called the *boot flow*, that takes place between the system power up and the beginning of the code of interest.

The boot flow is a multi-stage process during which different programs are responsible for initializing parts of the hardware and engaging the next booting stage [93]. The boot flow considered in our bare-metal application is shown in Fig. 4.14.

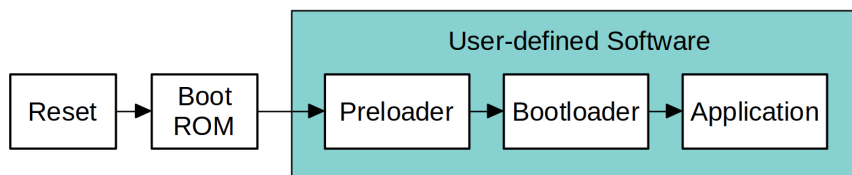


Figure 4.14: Boot flow of the HPS for a bare-metal application.

After powering up the processor, the first stage is the *Boot ROM* and is integrated inside the HPS. Only subsequent booting stages are provided by external tools and can be configured.

The Boot ROM program runs on the OCRAM its role is to enable the following hardware elements :

- Enable the L1 instruction cache, branch prediction, VFP and NEON units for CPU0 while CPU1 is held in reset.
- Configure the main clock.
- Configure the watchdog timer.
- Configure the flash controller.
- Locate and pass control to the preloader.

The watchdog timer role is to automatically reset the system after a certain duration. The boot ROM program copies the preloader program from an external memory source to the OCRAM and executes it. This external source is configurable with physical switches on the board, our application considers that all program elements are held in the SD/MMC flash memory card.

The preloader's role [93] is to configure the following elements :

- Configure the peripheral clocks.
- Invalidate the cache and branch predictor previously used by the boot ROM program.
- Initialize the SDRAM main memory.
- Remap the OCRAM to the bottom of the virtual address space. This region was previously pointing to boot ROM to use the vector table established by the boot ROM code. After the remap, the preloader vector table is used.
- Copy the bootloader to SDRAM.

As it is part of the user-specified software of the boot flow, the preloader must be configured by the programmer. It is built based on hardware descriptive files specific to the board as well as generic driver source codes. The preloader used in this work is based on the *Golden Hardware Reference Design* (GHRD) for the DE10-Nano Board. It is further configured to disable the watchdog timer and to load the next boot stage (bootloader) from the FAT partition of the SD/MMC.

The bootloader is the last stage of the boot flow and its role is therefore to load the target application to SDRAM. The bootloader selected in this work is the widely used open source *U-Boot* bootloader, built along with the preloader in the GHRD. It is normally designed to load the Linux OS but instead, it is configured here to load the application stored on the FAT partition via a U-Boot script.

4.6.3 SD/MMC partition table

When using the SD/MMC there exist two main ways of storing application related files.

- The card could be used in *raw* mode : files are copied by the programmer in binary format at addresses of its choice. This can easily become very complicated when several files must be copied and addresses must be manually tracked. It can further become a problem when specific files of the application must comply to a specific standards.
- The card could alternatively include a *partition table* which further describes one or more *file systems* on the card. A file system generally denotes a system that manages data on a storage device, facilitating file accesses and modifications. Partitions allow several file systems to coexist on a single storage device, each one of them having a defined location and size written in a partition table. This option is definitely preferable and is therefore considered in this instance.

The *Master Boot Record* (MBR) is a 512 bytes area at the very beginning of a memory storage, that holds the partition table of that storage. On the one hand, the boot ROM of the HPS requires the preloader to be located on the first partition of an MBR table with an *A2 partition ID* [101]. This partition must therefore be created on the SD/MMC in order for the preloader to be located by the boot ROM. On the other hand, a second partition needs to be added with a filesystem to handle the different files needed by the application.

The File Allocation Table (FAT) Filesystem has been chosen for the following reasons :

- U-Boot possesses specific functions to load an executable into SDRAM from a FAT filesystem.
- There exists lightweight open source libraries in C that handle reading and writing files into a FAT filesystem.
- The FAT16 variant is easy to understand although it is only capable of handling 32 MB filesystems.

The SD/MMC is therefore partitioned as illustrated in Fig. 4.15. The A2 partition for the preloader has a size of 1 MB and the second FAT16 partition has a size of 32 MB. The rest of card remains unallocated. The bootloader and U-Boot script are present in the FAT filesystem for initialization as well as the application executable and associated data file(s).

The target application uses the open source *FAT FS C* library for handling files from a FAT partition. It was chosen for its abundant documentation and its modularity. The library can be built to only make use of essential read functions, reducing its size to only a few kB of code size and runtime memory usage.

4.6.4 The GCC toolchain

To compile the target application, this work uses the *Sourcery CodeBench Lite* version of the *GNU Compiler Collection* (GCC) 6.2.0 [96], provided by *Mentor Graphics*. They are

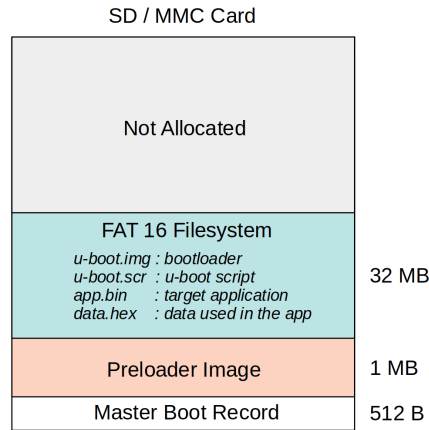


Figure 4.15: SD/MMC partitions and related files.

free softwares provided in the ARM Development Studio (DS-5) and comprises C and C++ compilers, assemblers, linkers, libraries and debugging tools [102]. The offered compiler offers features specific to ARM Altera processors, assumes a bare-metal operation and complies with the ARM Embedded Application Binary Interface EABI. Another popular choice of compilers for ARM targets is the ARM Compiler but it requires a license delivered by the full version of DS-5. Upon compilation, the compiler uses a variety of back-end tools that produces optimized code for different criteria depending on user-defined parameters. Since the number of parameters for the compiler is monumental, this small section aims at reviewing a few portions of those parameters that are essential for the present application. Those parameters are invoked when compiling `.c` and `.cpp` C/C++ source files into `.o` object files with the `gcc` or `g++` compilers.

In order to generate the best code for the Cortex-A9, the compiler and linker needs to get supplementary information about the processor. The following options are not specifically associated with any trade-off (execution time, produced file size, ...), they only allow the compiler to better know the available resources [96].

- `-march=armv7-a` specifies the target ARM architecture and defines the instruction set.
- `-mcpu=cortex-a9` specifies the target ARM processor. It is more specific than `-march` as it allows the compiler to optimally schedule instructions based on the processor's pipelines.
- `-mtune=cortex-a9` specifies the target ARM processor and optimizes for it, even with no `-mcpu` defined.
- `-mfloat-abi=softfp` specifies the ABI to be used when passing floating-point arguments to C/C++ functions. The `softfp` options passes them through ARM registers as for any other data-type. Alternatively, the `hard` option changes the ABI to pass them into NEON / VFP registers directly. Unfortunately, the linker does not allow object files with different ABIs to be linked together. Because the Sourcery CodeBench Lite does not provide GCC libraries that are compiled for the `hard` ABI, only the `softfp` option is available.
- `-mfpu=neon-fp16` specifies if the target possesses a hardware specific floating-point unit (FPU). The Cortex-A9 processor can choose either the NEON or VFPv3 unit.

However, when the NEON unit is specified, the compiler can only generate NEON floating-point instructions when additional optimization flags are included. This is because the NEON unit is not IEEE-754 compliant and therefore requires the `-f-unsafe-math-optimization` flag. If `fp16` is indicated, the compiler allows to use the `__fp16` data-type for storage of half precision floating-point numbers.

The compiler further accepts flags associated with authorized modifications of the source code by the compiler for optimization. These options are numerous and regrouped into different levels depending on their effect on the code size and execution time. Those levels are specified with the `-O<level>` flag. Without any optimization flag, the compiler produces a program whose instructions are directly related with C/C++ statements : any variable created in a function will result in an additional element present on its stack. This is useful for debugging in order to track the behavior of the program but suboptimal with regard to both the code size and execution time.

Using optimization flags makes the compilation time longer and produces a program that is potentially harder to debug. However, it can significantly improve its performance. The following list indicates the major modifications caused by different optimization levels [96].

- `-O0` is the default parameter and implies no optimization.
- `-O1` optimizes for both code size and execution time with minor source modifications such as reordering instructions and moving invariants out of loops.
- `-O2` comprises `-O1` flags and optimizes further by turning on every optimization flag that are not concerned with a code size / speed trade-off.
- `-O3` comprises `-O2` flags and turns on all optimization flags associated with a faster execution time, at the expense of code size. This includes the `-ftree-vectorize` flag that enables automatic vectorization into SIMD instructions.
- `-Os` comprises `-O2` flags and turns on all optimization flags associated with a smaller code size, at the expense of execution speed.
- `-Ofast` comprises `-O3` flags and pushes the execution speed by disregarding compliances with program standards. This includes the `-f-unsafe-math-optimization` flag that enables non-compliant IEEE-754 floating-point arithmetics. The compiler also reorders associative operations for faster execution regardless of rounding errors and disables checking of unauthorized math operations such as division by zero.

Finally, additional ARM specific flags are used in this work. They are regrouped separately since they can have an impact on the code size / speed trade-off.

- `-marm` specifies the compiler to use the ARM instruction set instead of the Thumb instruction set.
- `-mfp16-format=ieee` specifies the format of half precision floating-point numbers. The `alternative` option allows to use a broader range of numbers but does not support special cases such as infinite and NaN values. When using half precision floating-point numbers in this work, the IEEE standard is chosen to have easier conversion of data between the single and half precision formats.

- `-mno-unaligned-access` disables reading 32-bit memory words from addresses that are not 32-bit aligned. Throughout this work, it has been observed that enabling unaligned accesses can cause problems when combined with other optimization flags ; it is therefore deactivated.

4.6.5 Linkerscript and startup sequence

After compilation, the different object file needs to be combined to produce the binary executable, additional libraries of the C/C++ language are also added. This operation is performed by the *linker* and controlled by the *linker script* [96]. The linker script serves several purposes :

- It dictates the memory addresses for placement of different types of code (instructions, uninitialized data, initialized data, etc).
- It defines symbols for memory addresses that can be used inside the code.
- It provides a *startup* and *termination* code that executes before and after the target application's code (constructors, destructors, etc). It automatically fetches the default low-level library required by the language.
- It provides a set of interrupt vectors and handlers.
- It can optionally provide a *hosting* library offering I/O functions to use for debugging (`printf`, `open`, etc). When enabled, calls to those functions are translated into equivalent calls on the system that hosts the debugger.

The Common Startup Code Sequence (CS3) [102], particular to the Sourcery CodeBench release of GCC, provides adequate startup and termination code and basis linking scripts for the Cyclone V. The startup sequence is divided into 3 phases :

- The *hard reset phase* initializes the memory controllers for the board.
- The *assembly initialization phase* initializes the stack pointer.
- The *C initialization phase* initializes global and static variables, clears uninitialized data, constructs static C++ objects and calls the `main` function. When `main` returns, the termination code calls object destructors.

The linker script defines memory regions that reflects the memory structure of the board (Fig. 4.13) and different parts of the compiled code can be assigned to either one of those regions. This work do not leverage the advantage of the small OCRM memory as its capacity is well under the required memory size of the application. Instead the program uses the SDRAM in its entirety to store its instructions, data, stack and heap. The stack starts on the top address of the SDRAM and grows downwards while the heap starts at the end of the program code and grows upwards (Fig. 4.16). An additional buffer of 16 MB for the monitoring of the core by the Barman library is created in the linker script before the heap.

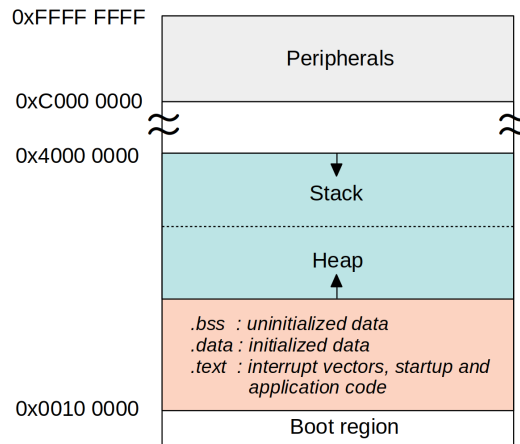


Figure 4.16: Code organization in the memory space of the Cyclone V.

4.6.6 Application initialization code

After the initialization steps performed in both the boot flow and the startup sequence, some hardware components still need to be activated or configured :

- Although the L1 instruction cache has been activated, the L1 data cache and L2 unified cache are still disabled.
- The cache policies need to be configured in the virtual table of the MMU.
- The SD/MMC FAT partition needs to be mounted in order to access the files contained in it.
- The PMU and L2 counters need to be configured to record core events.
- The UART controller should be enabled to display target information through a USB connection. Alternatively, hosted libraries could be used instead but would bring additional memory needs.

The remaining of this section therefore focuses on providing an adequate initialization sequence for those components. The proposed solution is adapted from the project examples provided by DS-5 and related Altera hardware libraries.

The cache inner and outer cache policy is configured by changing the memory region attributes in the MMU translation table. Setting a new page table would normally require accessing manually the CP15 coprocessor with Assembly instructions to modify the stored page table base address. Fortunately, Altera provides a *hardware library* in C language that brings a simpler interface. The page table is set with the following attributes :

- Addresses 0x0000 0000 to 0x4000 0000 that maps to the SDRAM are marked as cacheable with a write-back allocate inner and outer policy.
- Addresses 0xC000 0000 to 0xFFFF FFFF that maps to processor registers and peripherals are marked as non-cacheable. This is needed as those registers are *volatile* : their value might change independently of the processor's actions. A hardware counter is a great example of volatile data.

- Other addresses of the virtual memory space are marked as inaccessible. Any access to those addresses results in a hardware exception.

Since the *memory attributes* are modified on a 16 MB boundary, the page table is filled with 16 MB super-section descriptors. As explained in Appendix B.3, the use of super-sections generally results in better performances. The number of cycles necessary for address translation is lower due to better performances of the TLBs.

Hardware library functions automate the necessary processes upon a change of the translation table such as invalidation of the TLBs, caches and branch prediction.

Similarly, the cache hierarchy is activated along with the cache pre-fetcher that allows the use of the PLD instruction. The L2 monitoring bus is enabled to register events.

The hardware library does not implement PMU related operations, complementary functions are therefore written to complete the hardware library and configure the PMU counters through CP15.

At last, the SD/MMC is initialized and the 32 MB FAT partition is mounted. The lightweight FAT FS library is used and provides functions for mounting the drive, opening files and reading/writing operations. The FAT FS library is compatible with a wide range of processors and micro-controllers ; the interface is achieved by requiring the programmer to implement specific I/O functions. The hardware library of Altera already provides such function providing status information about the SD/MMC controller, reading and writing operations to addresses in the card.

4.6.7 Project organization

The complete bare-metal framework is illustrated in Fig. 4.17 and is the basis upon which the target application is constructed in the following chapter.

All support files and libraries are compiled with the `-Os` flag for size economy. Since their use is mainly during the initialization phase, the trade-off with speed can be ignored. On the other hand, the optimized program of the target application is compiled for best performances with `-Ofast` flag.

4.6.8 Summary

Optimization flags of the compiler and memory configuration parameters for the bare-metal framework are summarized in Table 4.11.

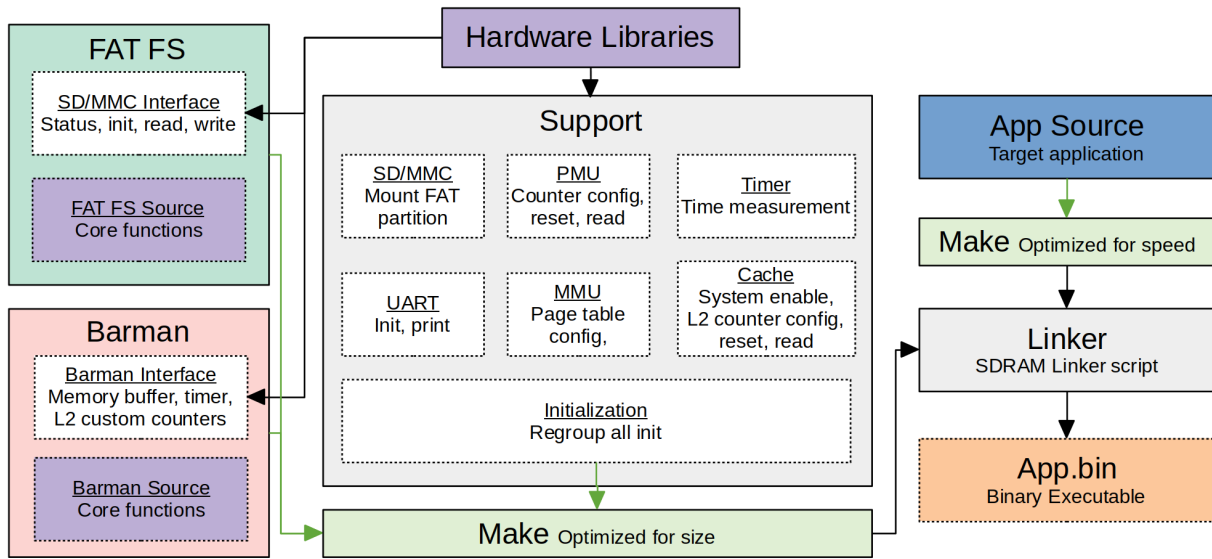


Figure 4.17: Bare-metal Framework for the Cyclone V.

Global GCC flags	L1 Cache features	MMU features
-marm -march=armv7-a -mcpu=cortex-a9 -mtune=cortex-a9 -mfloat-abi=softfp -mfpu=neon-fp16 -mfp16-format=ieee -mno-unaligned-access SDRAM linker-script	L1 Instruction cache enabled L1 Data cache enabled Branch prediction enabled L1 Pre-fetch buffer enabled	MMU enabled Inner / outer write-back allocate policy on the 1 GB memory space of SDRAM 16 MB super-section translation table entries.
Support files GCC flags	Application files GCC flags	L2 Cache features
-Os	-Ofast	L2 Cache enabled L2 Pre-fetch buffer enabled

Table 4.11: Bare-metal Framework features.

CHAPTER 5

OPTIMIZATION OF CONVOLUTIONAL SPIKING NEURAL NETWORKS ON THE CORTEX-A9 PROCESSOR

After the complete overview of the Cortex-A9 features, the goal set in this chapter is to create a real-time algorithm for the Slayer model optimized on the N-CARS dataset. The first part of this chapter focuses on a reformulation of the Slayer constitutive equations in order to save computations and allow to directly process a stream of events coming from an event-based camera. In addition, the large sparsity observed in the spike signals between layers is leveraged by formulating event-based convolutional and fully-connected layer operations. A complexity comparison with the original algorithm is provided to motivate the proposed approach.

The algorithms are then integrated in the C/C++ Baremetal framework established in the last chapter and each method is implemented in order to leverage the computational efficiency of NEON SIMD instructions and the cache hierarchies. The solution is profiled several times with incremental changes to the algorithm and memory organization in order to produce the best performances. A complete benchmarking is then conducted on the best solution to measure the execution time in terms of the number of spikes and the layer sizes. It provides essential metrics to further optimize the model hyper-parameters with the target architecture in mind for future works. The single and half floating-point data-types are considered in this approach and yielded convincing results thanks to the specialized NEON hardware.

Finally, the program is simulated with samples of the N-CARS dataset in order to validate the obtained metrics in the benchmarks and assess if real-time constraints are met. Obtained latencies are then compared with state-of-the-art techniques on the N-CARS dataset and demonstrated competitive performances.

5.1 Slayer real-time reformulation

Chapter 3 described in details the operations behind the Slayer SNN. The Pytorch framework processes samples arranged in 4D tensors with axes for the channel, height, width and time dimensions.

- The model does not benefit from the event-based sparse nature of the spike signals of the input and hidden layers. Therefore the algorithm’s complexity for the forward pass does not scale with the input activity.
- The algorithm has to wait for the total length of the sample to be available. If connected with the DVS, several milliseconds have to elapse before the sample is ready. Moreover, the entire sample needs to be stored in memory before processing. The induced latencies could prevent the deployment of the model in a fast real-time application such as autonomous driving.

Nevertheless this structure is logical when considering the back-propagation. Indeed, Equations 3.23 and 3.25 underline that the full length of the sample is necessary to compute the gradients. Furthermore, the Pytorch Framework is supported by CUDA kernels that allow to delegate the processing to a GPU, with a high degree of parallelism.

The goal of this reframing is therefore to provide a novel algorithm that would exclusively operate on a CPU and in a real-time environment in order to target embedded systems. It would operate at the sampling period of the model, be agnostic of the total sample length, and process directly the data coming from a DVS camera. Furthermore, it would be optimized for a specific model in forward operation exclusively. The benefits are multiple :

- The network is connected to a DVS camera and outputs spikes at sampling rate of the model. For the optimized model found in Section 3.4.4, this corresponds to a 2 ms latency. This can allow a supervisor to monitor the output spikes and decide when to stop the model in order to achieve an accuracy/latency trade-off.
- A real-time approach implies processing the incoming events of a single time-step. It has the potential to lower the complexity of a layer operation because of the sparse nature of spike signals. Convolutional and dense operations can be adapted to account for the high degree of sparsity. On the other hand, this formulation introduces tight real-time constraints : each event of a given time-step must be processed before the events of the next time-step are received. Also, the model therefore needs a form of *state memory* to account for the processing realized in previous time-steps.
- Optimizing for a particular model allows for different organization of the software. Additionally, restricting the model to operate solely in the forward operation for inference allows to remove all the maintenance operations of the forward pass necessary for the backward pass (storage of intermediate activations, etc).

The following two sections are therefore organized as follows. Section 5.1.1 reformulates the neuron model of Slayer in order to generate output spikes in real-time, one time-step

at a time. Sections 5.1.2 and 5.1.3 explores how sparsity can be exploited in the dense and convolutional operations.

The following nomenclature is used in the next section :

- The convolution and dense operations are referred as the *layer operation*.
- The spike filtering operated on the input spike signal and thresholding of the membrane potential that produces the output spike signal are together referred as the *update operation*. This denomination becomes apparent in the next section.

5.1.1 Infinite impulse response model

In order to compute the membrane potential of a neuron and its associated output spike sequentially, one time-step after another, an *infinite impulse response* (IIR) filter formulation can be used. It implies first expressing the filter and input signals in a discrete time domain and secondly applying the *Z-Transform* to obtain the filter coefficients. This formulation therefore allows to express the new membrane potential in terms of previously stored states thanks to an *update equation*. Particular care should be taken in the analytical derivations when considering the retroaction that exists between the membrane potential and the output spike signal in order to avoid a positive feedback. In addition, the required storage for filter's states should be minimal.

5.1.1.1 Formulation

Starting from the sampled formulation of Slayer previously introduced in Eq. 3.6, the membrane potential signal $\mathbf{u}^{(l+1)}[n]$ of neurons in layer (l) is expressed in Eq. 5.1 in terms of a linear function of the spike response signals $\mathbf{a}^{(l)}[n]$ and the refractory response $\mathbf{r}^{(l+1)}[n]$. The linear function is parametrized by the weight tensor $\mathbf{W}^{(l)}$. The spike response signal and the refractory response are expressed in Eq. 5.2 and 5.3 ; they are computed via a temporal convolution with filters $\epsilon[n]$ and $\nu[n]$ respectively. A new spike is produced at time-step n if the membrane potential crosses the threshold θ , as described by the spike function $f_s(u)$ in Eq. 5.4.

$$\mathbf{u}^{(l+1)}[n] = f\left(\mathbf{a}^{(l)}[n], \mathbf{W}^{(l)}\right) + \mathbf{r}^{(l+1)}[n] \quad (5.1)$$

$$\mathbf{a}^{(l)}[n] = \left(\epsilon \circledast \mathbf{s}^{(l)}\right)[n] \quad (5.2)$$

$$\mathbf{r}^{(l+1)}[n] = \left(\nu \circledast \mathbf{s}^{(l+1)}\right)[n] \quad (5.3)$$

$$f_s(u) : u \rightarrow s, s[n] := s[n] + \delta(n - n_{f+1}) \text{ where } n_{f+1} = \min\{n : u[n] = \theta, n > n_f\} \quad (5.4)$$

As already seen in Eq. 3.8, both kernels are defined as a product between a linear and an exponential. With a change of variable for sampling, the filters have therefore the expressions shown in Eq. 5.5

$$n := \frac{t}{T_s}$$

$$\epsilon[n] = \frac{nT_s}{\tau_s} \exp\left(1 - \frac{nT_s}{\tau_s}\right) \Theta[n] \quad \nu[n] = -S\theta \frac{nT_s}{\tau_r} \exp\left(1 - \frac{nT_s}{\tau_r}\right) \Theta[n] \quad (5.5)$$

where $\Theta[n]$ is the step function

$$\Theta[n] = \begin{cases} 1 & \text{if } n \geq 0 \\ 0 & \text{otherwise} \end{cases} \quad (5.6)$$

As both filters belong to the same family of functions, their formulas can be generalized as $h[n]$ in Eq. 5.7 with different coefficients.

$$h[n] := Cna^n \quad \text{with} \quad \begin{cases} C_\epsilon = \frac{T_s e}{\tau_s}, & a_\epsilon = e^{-\frac{T_s}{\tau_s}} \\ C_\nu = \frac{-S\theta T_s e}{\tau_r}, & a_\nu = e^{-\frac{T_s}{\tau_r}} \end{cases} \quad (5.7)$$

The *Z-Transform* of the discrete filter $h[n]$ results in the following second order transfer function.

$$H(z) := \mathcal{Z}\{h[n]\} = \frac{Caz^{-1}}{(1 - az^{-1})^2} = \frac{Caz^{-1}}{1 - 2az^{-1} + a^2z^{-2}} \quad (5.8)$$

The membrane equation is written again in terms of the two filter's transfer function in Eq. 5.9. Thanks to the linear property of both the dense and the convolution operation, the temporal convolution and the layer operation can be swapped. This formulation is referred below as the *alternate Slayer formulation* as will unlock the potential of adaption the layer operation f for sparse inputs. The signal $\mathbf{x}^{(l)}[n]$ is called the *filter input* or also the *feature map* in the case of the convolution operation.

$$\mathbf{u}^{(l+1)}[n] = \epsilon(z) f \underbrace{\left(\mathbf{s}^{(l)}[n], \mathbf{W}^{(l)}\right)}_{\mathbf{x}^{(l)}[n]} + \nu(z) \mathbf{s}^{(l+1)}[n] \quad (5.9)$$

The diagram of Fig. 5.1 illustrates the relations between the filter input, the membrane potential and the output spikes for a single neuron.

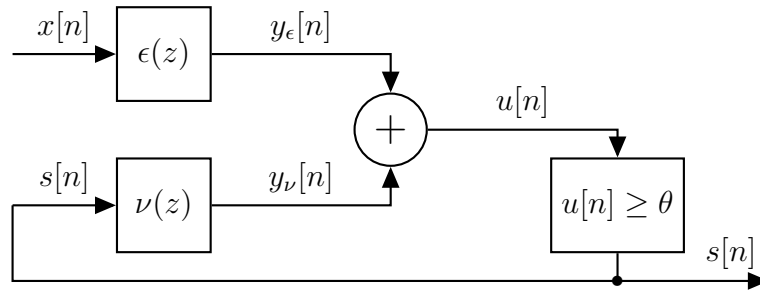


Figure 5.1: Neuron Update Block Diagram.

The signal $x[n]$ is filtered by the transfer function (TF) $\epsilon(z)$ and then added with the refractory response coming from the TF $\nu(z)$. The obtained membrane potential $u[n]$ is then compared with θ to produce a binary output spike signal $s[n]$ whose value is either 0 or 1. The feedback loop is readily visible as the output spike signal is recirculated to

generate the refractory response. This would normally cause the state to be undetermined ; however, as seen in Section 3.2.1, the internal structure of the filter prevents this situation from happening. The following calculations describes the internal representation of the two filters.

Recalling TF of Eq. 5.8, the Z-Transform of the filtered signal can be written as

$$\begin{aligned} Y(z) &= H(z)X(z) \\ \Leftrightarrow (1 - 2az^{-1} + a^2z^{-2})Y(z) &= Cax^{-1}X(z). \end{aligned}$$

Taking the Inverse Z-Transform on both sides leaves

$$\begin{aligned} \Leftrightarrow y[n] - 2ay[n-1] + a^2y[n-2] &= Cax[n-1] \\ \Leftrightarrow y[n] &= Cax[n-1] + 2ay[n-1] - a^2y[n-2]. \end{aligned} \quad (5.10)$$

At first sight, the difference equation of Eq. 5.10 requires 3 states for the IIR filter. This form, called *Direct Form I* is not *canonical with respect to delay* as it is always possible to implement an N th-order filter using only N delay elements [103]. Commutation rules of Linear Time-invariant (LTI) systems, illustrated in Fig. 5.2 for a generic second order filter, allow to obtain the transposed *Direct Form II* which is *canonical is terms of delay* for a second order difference equation of the form

$$y[n] = b_0x[n] + b_1x[n-1] + b_2x[n-2] - a_1y[n-1] - a_2y[n-2].$$

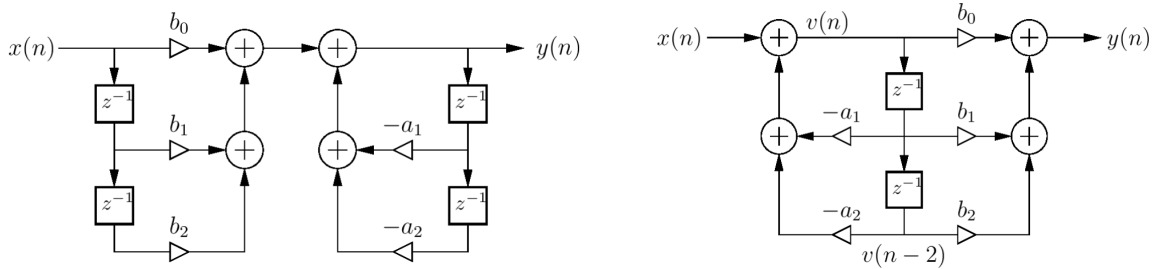


Figure 5.2: IIR Direct Forms I and II, reproduced from [103]

Applying those principles to $H(z)$ gives the block diagram of Fig. 5.3 with coefficients f_1, b_1, b_2 . Only two states are necessary and the output is not directly related with the input, allowing for a closed-loop operation.

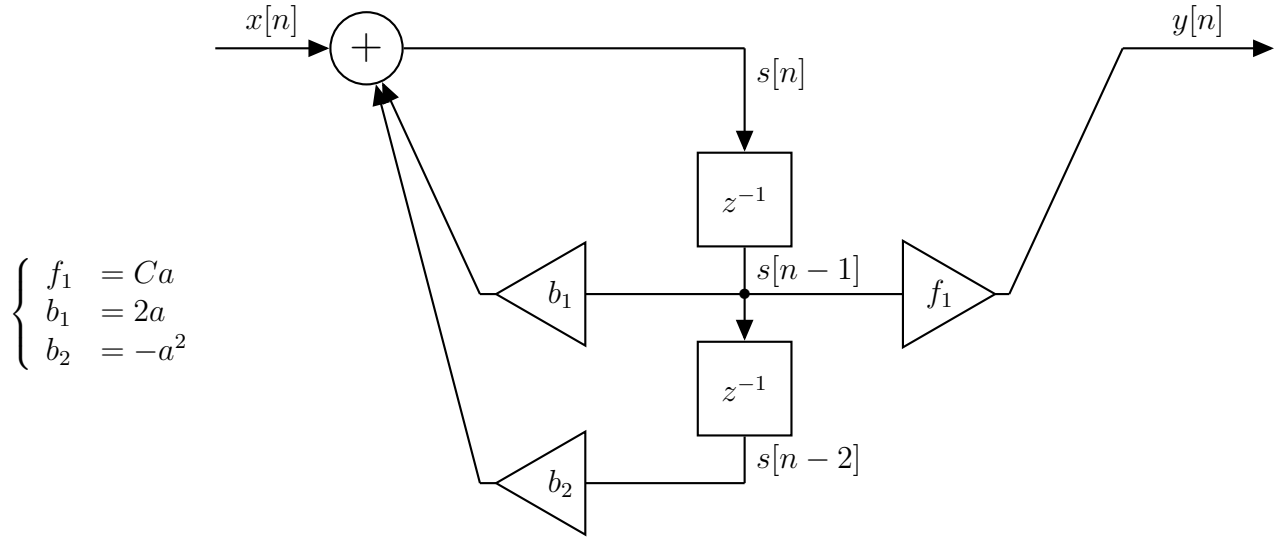


Figure 5.3: IIR Filter Block Diagram

5.1.1.2 Algorithm

The following snippet in Fig. 5.4, in C++, establishes the basis for the *update algorithm* for all neurons of a layer. It is expressed by combining Fig. 5.1 and 5.3.

The function takes as parameters arrays of length `len` where each entry is associated with a different neuron of a layer. Parameters are :

1. The filter input.
2. Each state of the two filters.
3. The length of the input arrays.
4. The output array that collects the generated spikes.

First, the output spike is generated from the sum of the output of both filters and is compared to `theta`. If there is an output spike, the index of the neuron that is being updated `n` is stored in an `output` array. Then the new state `s0` corresponding to $s[n]$ of Fig. 5.3 is computed and finally all states are shifted. The length of the `output` array is returned.

This algorithm is far from optimal in terms of memory usage, cache usage and potential use of processor's specialized hardware. This results in a rather slow algorithm that is going to need further improvements through cautious memory rearrangements and the use of NEON SIMD intrinsic function in C++, available for the Cortex-A9. Those aspects are developed in Section 5.2.

5.1.1.3 Equivalence of the two methods

A comparison with the FIR filter of the original Slayer implementation is necessary to assess the equivalence. To remind the reader, the filtering in the Pytorch implementation

```

1 unsigned int update_state(
2     float *input,
3     float *s1_eps, float *s2_eps, float *s1_nu, float *s2_nu,
4     const float f1_eps, const float f1_nu,
5     const float b1_eps, const float b1_nu,
6     const float b2_eps, const float b2_nu,
7     const float theta,
8     unsigned int *output, const unsigned int len)
9 {
10     // Number of output spikes
11     unsigned int len_output = 0;
12
13     // For each Neuron in the layer
14     for(unsigned int n{0}; n < len; ++n)
15     {
16         // if u[n] >= theta, save the output spike
17         bool spike = ((f1_eps * s1_eps[n] + f1_nu * s1_nu[n]) >= theta);
18         if (spike)
19             output[len_output++] = n;
20
21         // Epsilon filter
22         float s0_eps = input[n] + b1_eps * s1_eps[n] + b1_nu * s2_eps[n];
23
24         // Nu filter
25         float s0_nu = ((float) spike) + b1_nu * s1_nu[n] + b2_nu * s2_nu[n];
26
27         // Save states
28         s2_eps[n] = s1_eps[n];
29         s2_nu[n] = s1_nu[n];
30         s1_eps[n] = s0_eps;
31         s1_nu[n] = s0_nu;
32     }
33
34     return len_output;
35 }

```

Figure 5.4: Basic IIR update rule

is computed via an FIR convolution operation on a GPU based on the entire duration of the input sample and a truncated impulse response of the filter. The filter lengths for the optimized model are 28 and 18 for the ϵ and ν filters respectively, with a sample period $T_s = 2$ [ms]. As observed in Fig. 5.5, the two methods leave equivalent results for the same input spike signal.

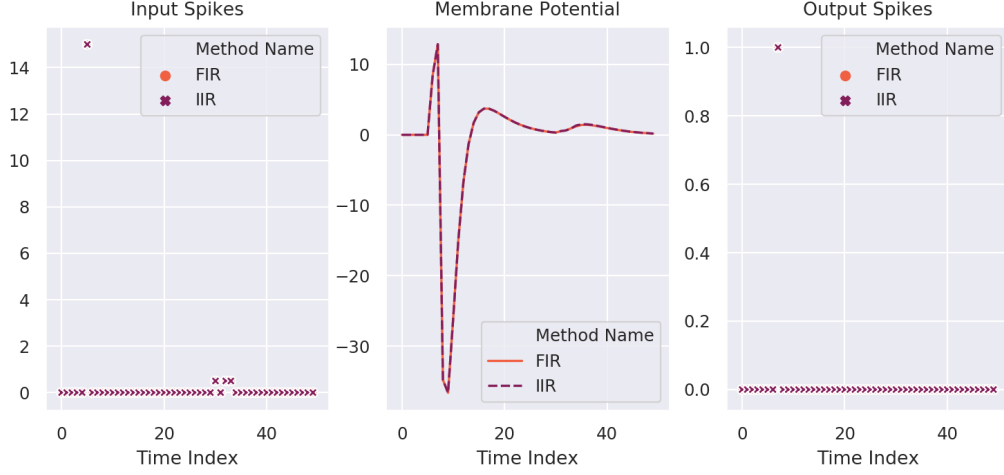
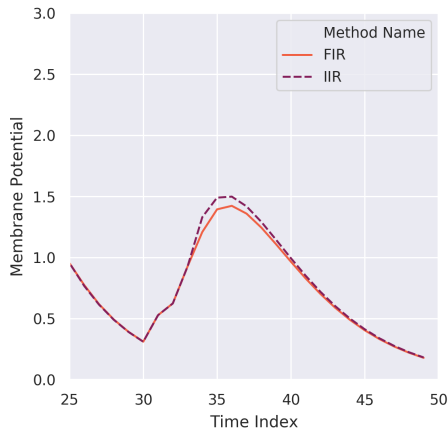
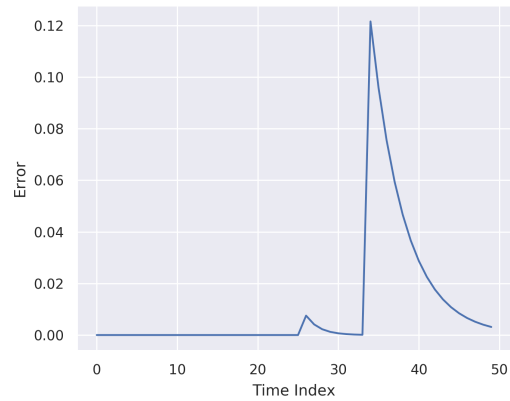


Figure 5.5: FIR and IIR comparison.

However, a careful examination highlights the effects of the filter truncation in the original implementation, as seen in Fig. 5.6a. As the first non-zero input occurs at discrete index $n = 15$, the first noticeable truncation error appears 18 steps later at index $n = 33$. A graph of the error between the two methods reveals the missing part of the filter (Fig. 5.6b). A solution to this problem is to increase the filter length in the Pytorch implementation to reduce the truncation error at the cost of an increase in execution time, the ideal case being that filter length is at minimum equal to the sample duration. Instead, we choose here to keep the Pytorch implementation as it is and particular attention is taken to track possible changes in model accuracy.



(a) FIR and IIR comparison magnified.



(b) Filters truncation error.

5.1.1.4 Complexity comparison

To conclude this section, a comparison of the algorithmic complexity between the FIR and IIR complexity is appropriate. The proposed method to compare the two algorithms is to count the number of Floating-point Operations (FLOP) ; it provides a first order approximation of the execution time if both algorithms were executed on the same platform. However this metric is based on several assumptions :

- Floating-point operations are dominant in the algorithm and integer operations execution time is negligible in comparison.
- Floating-point instructions take a similar number of cycle to execute. This is a very strong assumption since in practice **VMUL** instructions take more cycles than **VADD** and **VSUB**. Moreover, a floating-point division does not have a dedicated hardware and relies on software libraries, considerably lengthening the number of cycles for this operation.
- Memory access time, parallel processing and SIMD operations are not taken into account.

IIR complexity As explained in Section 3.3.1, the Pytorch implementation generates the output spike in two steps. First of all, the PSP of the neuron is generated from the input spike signal. Then the refractory response is added to the result of the layer operation and the output spike signal is generated. There are therefore two IIR operations :

1. The input spike signal $s_i^{(l)}(t)$ is convolved with the ϵ kernel.
2. When $x_i^{(l)}(t)$ reaches the threshold θ , the ν kernel is added to subsequent time-step.

The FIR filter algorithm can therefore be written as in Fig. 5.7

Each iteration of the **psp** function includes one floating-point addition and one floating-point multiplication. Accounting for the border effects introduced by the **if** clause, the total number of FLOP therefore is given by

$$\text{FLOP}_{\text{FIR PSP}}(N_s, N_\epsilon) := 2 \cdot \left(N_s \cdot N_\epsilon - \frac{N_\epsilon(N_\epsilon - 1)}{2} \right) \quad (5.11)$$

- N_s is the signal duration.
- N_ϵ is the filter size.

The FIR method additionally requires a separate method after the layer operation to generate the output spikes. The complexity here depends on the number of output spikes produced N_{spike} . However, since the complexity also depends on the location of the output spikes due to the **(n + i < N_s)** clause, the worst case is assumed. Each iteration has a floating-point comparison and depending on the result, several floating-point additions ensue.

$$\text{FLOP}_{\text{FIR Spike}}(N_s, N_\nu, N_{\text{spike}}) := N_s + N_{\text{spike}} \cdot N_\nu \quad (5.12)$$

```

1 void fir_psp(float *u, float *s_i)
2 {
3     for (unsigned int n{0}; n < N_s; ++n)
4         for (unsigned int i{0}; i < N_eps; ++i)
5             {
6                 int k = n - i;
7                 if (k >= 0)
8                     u[k] = s_i[k] * epsilon[i];
9             }
10 }
11
12 void fir_spike(float *x, float *s_o)
13 {
14     for (unsigned int n{0}; n < N_s; ++n)
15         if (x[n] >= theta)
16             {
17                 s_o[n] = 1;
18                 for (unsigned int i{0}; (i < N_nu) && (n + i < N_s); ++i)
19                     x[n+i] += nu[i];
20             }
21 }

```

Figure 5.7: FIR update rule

FIR complexity Based on the algorithm of Fig. 5.4, the main loop includes the following floating-point operations :

- 6 floating-point multiplications.
- 5 floating-point additions.
- 1 floating-point comparison.

The provided algorithm realizes one step of the update rule for all neurons of a layer. Instead we should multiply the number of FLOP for a single neuron by the number of time-steps. We therefore obtain a complexity that depends only on the number of time-steps.

$$\text{FLOP}_{\text{IIR}}(N_s) := 12 \cdot N_s \quad (5.13)$$

Comparison The two methods do not have the same parameters and therefore need the following assumptions for comparison :

1. The impact of the output spikes in the FIR formulation are ignored. One should therefore bear in mind that the comparison is optimistic for the FIR algorithm.
2. We only account for filter lengths that are smaller than the signal length.

The difference of the number of FLOP between the FIR and IIR formulations is illustrated in Fig. 5.8 for different values of signal and filter lengths. Negative values therefore indicate an FIR algorithm that performs better than the IIR. Positive values assess the contrary.

The result of this method shows the strong advantage of the IIR formulation for ϵ filter sizes that exceeds 6 taps. Since the filter length of the optimized model is equal to 28 (Section 5.1.1.3), the IIR filter outperforms its FIR counterpart.

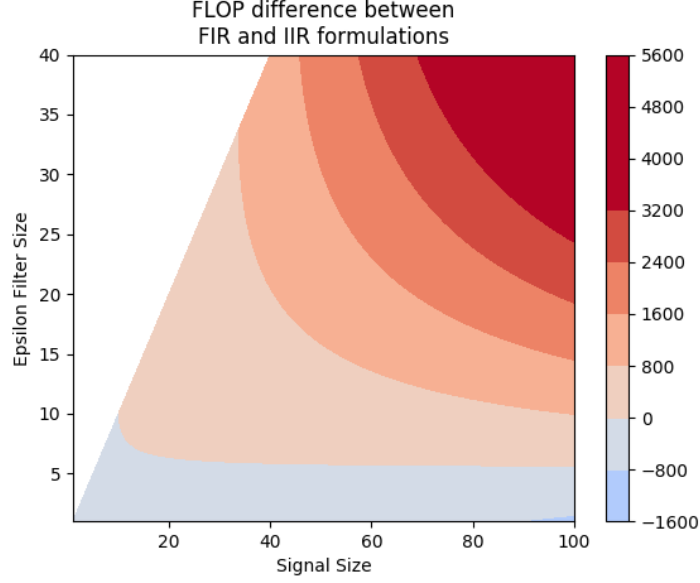


Figure 5.8: FIR and IIR FLOP Comparison.

On the other hand, the choice of a FIR implementation in the Pytorch frameworks makes sense since :

1. This algorithm benefits greatly from CUDA parallelization on a GPU.
2. Convolution operations are necessary for the forward pass while cross-correlation operations are required for the backward pass and gradient descent. The same FIR filter is used in both cases.

Now that the model for the individual time updates is established, the remaining two sections focus on simplifications one can bring to the dense and convolution operations in the context of sparse inputs. This assumption is backed by the analysis of Section 3.4.4 that reveals that the optimized network has sparse spike signals in-between layers (0-20%), especially for the convolutional layers (0-8%).

5.1.2 Event-based dense layer

Back in Eq. 1.1, the dense operation was introduced. In the particular case of the alternate Slayer formulation of Eq. 5.9, the input spike signal $\mathbf{s}^{(l)}[n]$ is multiplied by the weight matrix $\mathbf{W}^{(l)}$. This is reminded in Eq. 5.14 ; note that the Slayer formulation does not include biases.

$$x_i^{(l+1)}[n] = \sum_{k=0}^{N^{(l)}-1} s_k^{(l)}[n] W_{k,i}^{(l)} \quad (5.14)$$

For a given time-step n , the vector of input spikes is assumed to be sparse. Calling $\mathcal{E}_n^{(l)}$ the set of indices corresponding to non-zero values of the input vector, we obtain

$$s_k^{(l)}[n] = \begin{cases} 1 & \text{if } k \in \mathcal{E}_n^{(l)} \\ 0 & \text{otherwise} \end{cases} \quad (5.15)$$

Therefore, the layer operation can be simplified to only account for the indices in the set $\mathcal{E}_n^{(l)}$; this is given by

$$x_i^{(l+1)}[n] = \sum_{k \in \mathcal{E}_n^{(l)}} W_{k,i}^{(l)}. \quad (5.16)$$

Compared to a dense operation realized on the complete input, this algorithm has a complexity significantly lower. It avoids floating-point multiply instructions and the number of operations is a function of the *input activity*. In other terms, the computational time can be faster for highly sparse inputs. Denoting the *normal* and *sparse* algorithms, the complexity of the dense operation for all neurons of the layer is written as :

$$\text{FLOP}_{\text{Dense Normal}} = 2 \cdot N^{(l)} \cdot N^{(l+1)} \quad (5.17)$$

$$\text{FLOP}_{\text{Dense Sparse}} = |\mathcal{E}_n^{(l)}| \cdot N^{(l+1)} \quad (5.18)$$

where $|\mathcal{E}_n^{(l)}|$ is at most equal to $N^{(l)}$.

The C++ snippet of Fig. 5.9 implements the core of Eq. 5.16. The weight matrix is transposed with dimensions $N^{(l)} \times N^{(l+1)}$ and is contained in a flattened 1D array `weight`. Input events of time-step n are contained in the `array_spike` array of length `len_spike`. The dense operation fills the initialized array `input` that is later used as the input of the IIR filter.

```

1 void dense_operation(float *input, const float *weight, unsigned int *array_spike, const ←
   unsigned int len_layer, const unsigned int len_spike)
2 {
3     for (unsigned int event{0}; event < len_spike; ++event)
4     {
5         const float *rweight{weight + (len_layer * array_spike[event])};
6         for (unsigned int i{0}; i < len_layer; ++i)
7             input[i] += rweight[i];
8     }
9 }

```

Figure 5.9: Sparse dense layer operation.

Optimization of this function is considered in Section 5.2.6.1.

5.1.3 Event-based convolutional layer

The convolution operation for the alternate Slayer formulation is written in Eq. 5.19. As for the dense operation, the convolution operation complexity can be greatly reduced when leveraging the sparsity of the input.

$$\mathbf{X}_{c_o}^{(l)}[n] = \sum_{c_i=0}^{C^{(l)}-1} \mathcal{S}_{c_i}^{(l)}[n] \circledast \mathbf{W}_{c_i}^{(l)} \quad (5.19)$$

In order to ease computations, we will first optimize a 1D convolution operation in order to get the main insights necessary. Then the obtained results will be transposed to the multi-channel 2D convolution of the standard CNN. Also, the following parameters present in the

optimized model of Section 3.4.4 are assumed in order to take benefit of every mathematical property :

1. The stride size is equal to 2.
2. The input size N is odd.
3. The filter size K is odd.
4. The *valid* convolution is used.

5.1.3.1 Event-based 1D convolution

The 1D convolution operation is written in Eq. 5.20 for a generic input vector $\mathbf{x}$, a filter $\mathbf{w}$ and an output vector $\mathbf{y}$.

$$y_h = \sum_{k=0}^{K-1} w_k x_{2h-k+(K-1)} \quad (5.20)$$

- An additional offset $(K - 1)$ is necessary in the valid convolution.
- The output size is equal to $((L - K)/2 + 1)$.

Assuming that the input vector is sparse and binary, we have the following interesting property

$$x_h = \begin{cases} 1 & \text{if } h \in \mathcal{E} \\ 0 & \text{otherwise} \end{cases} \quad (5.21)$$

Therefore, if there is a single index in the event set $\mathcal{E}$, the following holds

$$y_h = w_k x_{h^*} \quad \text{for } h^* \in \mathcal{E} \text{ and } k \in [0, K[\quad (5.22)$$

h and k are linked to h^* via the relation

$$\begin{aligned} h^* &= 2h - k + (K - 1) \\ \iff h &= \frac{h^* + k - (K - 1)}{2} \end{aligned} \quad (5.23)$$

Since h and k indices, they must obey to the following constraints :

$$h \text{ and } k \text{ must be integers.} \quad (5.24)$$

$$k \in [0, K[\quad (5.25)$$

$$h \in [0, (L - K)/2 + 1[. \quad (5.26)$$

Since we assumed an odd filter length, $(K - 1)$ is even. Therefore Eq. 5.23 leaves the following conditions on k in order for Eq. 5.24 to be respected :

1. if h^* is odd, k must be odd.

2. if h^* is even, k must be even.

The kernel and input indices are linked together with the output indices based on their parity. In fact, a convolution with a stride size of 2 can be divided into two smaller convolutions involving the even and odd parts of the input and filters [104]. The obtained vectors are then summed together to obtain the output of the strided convolution. This property is illustrated in Fig. 5.10

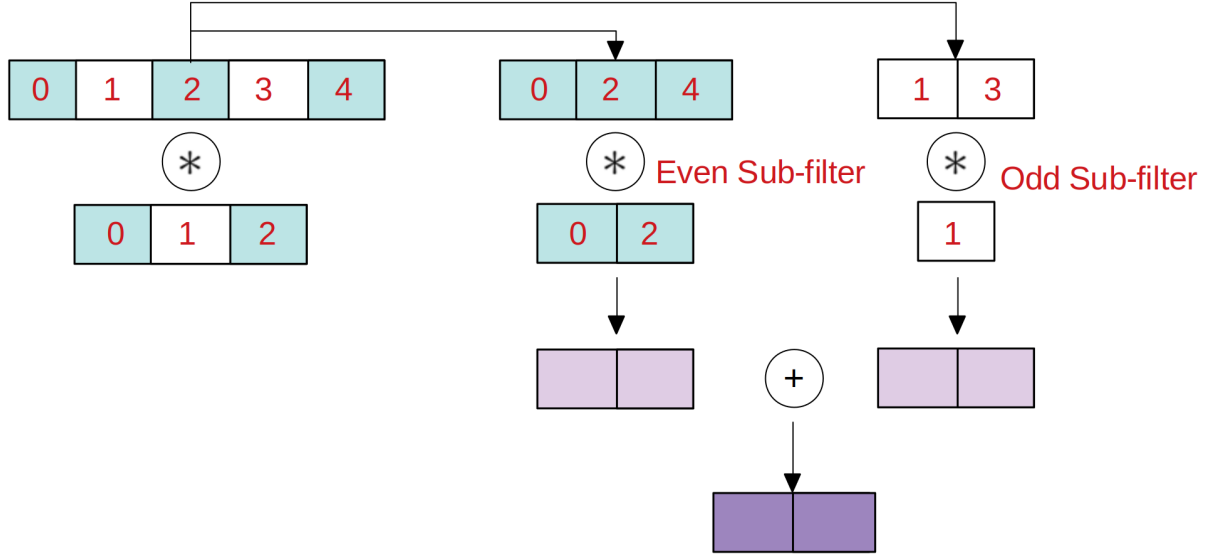


Figure 5.10: Even and odd filter separation in stride 2 convolution.

This means that in practice, the even and odd parts of a filter (*sub-filters*) can be stored separately because for a given event, only one sub-filter is used at a time. This further allows to have sequential indices that span the entries of the sub-filters in order to have better performances regarding memory accesses. This is further exploited in Section 5.2.1.2.

The goal now is therefore to express the border constraints of h (Eq. 5.26) on $\frac{k}{2}$ and $\frac{k-1}{2}$ to index the even and odd parts of the filter separately, as described in Eq. 5.27.

$$w_k = \begin{cases} w_k^{\text{even}} \\ w_{\frac{k-1}{2}}^{\text{odd}} \end{cases} \quad (5.27)$$

Even part

The condition for the lower bound yields

$$\begin{aligned} \frac{h^* + k - (K - 1)}{2} &\geq 0 \\ \Leftrightarrow \frac{k}{2} &\geq \frac{K - 1}{2} - \frac{h^*}{2} \end{aligned} \quad (5.28)$$

while the condition for the upper bound gives

$$\begin{aligned} \frac{h^* + k - (K - 1)}{2} &< \frac{L - K}{2} + 1 \\ \iff \frac{k}{2} &< \frac{L + 1}{2} - \frac{h^*}{2} \end{aligned} \quad (5.29)$$

Combined with Eq. 5.25, this yields the following bounds :

$$\frac{k}{2} \in \left[\max \left(0, \frac{K - 1}{2} - \frac{h^*}{2} \right), \min \left(\frac{K + 1}{2}, \frac{L + 1}{2} - \frac{h^*}{2} \right) \right]. \quad (5.30)$$

Odd part

Similarly, the condition for the lower bound yields

$$\begin{aligned} \frac{h^* + k - (K - 1)}{2} &\geq 0 \\ \iff \frac{k - 1}{2} &\geq \frac{K - 1}{2} - \frac{h^* + 1}{2} \end{aligned} \quad (5.31)$$

while the condition for the upper bound gives

$$\begin{aligned} \frac{h^* + k - (K - 1)}{2} &< \frac{L - K}{2} + 1 \\ \iff \frac{k - 1}{2} &< \frac{L + 1}{2} - \frac{h^* + 1}{2} \end{aligned} \quad (5.32)$$

Combined with Eq. 5.25, this yields the following bounds :

$$\frac{k - 1}{2} \in \left[\max \left(0, \frac{K - 1}{2} - \frac{h^* + 1}{2} \right), \min \left(\frac{K - 1}{2}, \frac{L + 1}{2} - \frac{h^* + 1}{2} \right) \right]. \quad (5.33)$$

Unification The two situations have a few invariants in common that allow to simplify the results.

$$\hat{k} := \left\lfloor \frac{k}{2} \right\rfloor \quad (5.34)$$

$$\hat{h}^* := \left\lfloor \frac{h^* + 1}{2} \right\rfloor \quad (5.35)$$

$$R_{\text{left}} := \frac{K - 1}{2} - \hat{h}^* \quad (5.36)$$

$$R_{\text{right}} := \frac{L + 1}{2} - \hat{h}^* \quad (5.37)$$

$$\hat{k} \in \left[\max(0, R_{\text{left}}), \min \left(\frac{K + 1}{2}, R_{\text{right}} \right) \right] \quad \text{if } h^* \text{ is even} \quad (5.38)$$

$$\hat{k} \in \left[\max(0, R_{\text{left}}), \min \left(\frac{K - 1}{2}, R_{\text{right}} \right) \right] \quad \text{if } h^* \text{ is odd} \quad (5.39)$$

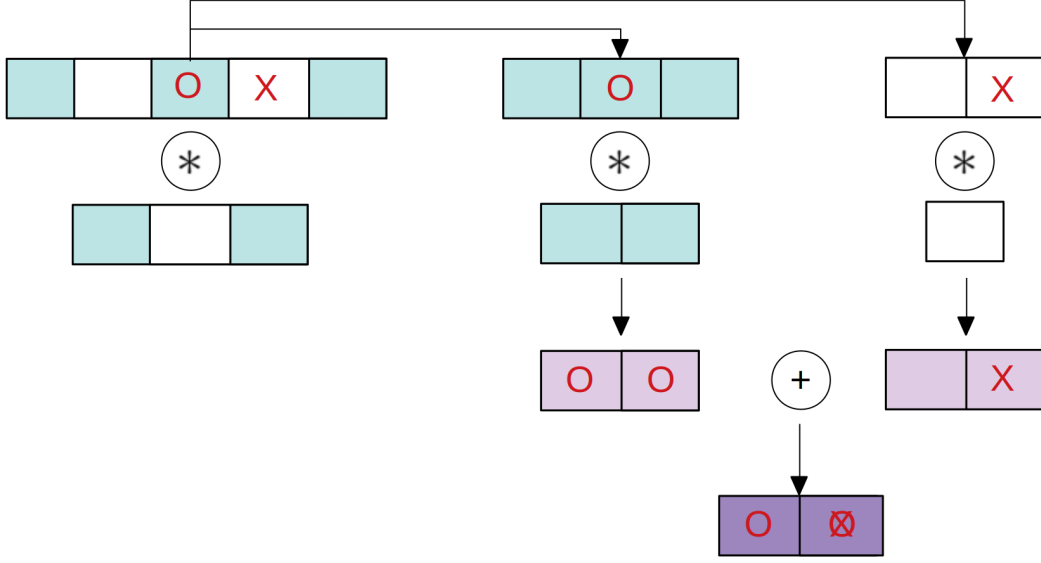


Figure 5.11: Overlap of outputs in 1D convolution. Red markers indicate separate events.

Reverting to the initial assumption in Eq. 5.22, it is obvious that in practice, a sparse vector will contain more than one non-zero value. Therefore, the established method can be used provided that the results in y_h are accumulated in the vector. In fact, if input events are next to each other, the associated outputs overlap ; this is illustrated in Fig. 5.11.

The pseudo-code of Fig. 5.12 illustrates the 1D event-based convolution. The quantities of Eq. 5.34 to 5.39 are computed for each input event.

Input: $\mathcal{E}^{(l)}$, w , K , $N^{(l)}$

Output: x

- 1: **for** (every event index $h^* \in \mathcal{E}^{(l)}$) **do**
- 2: From h^* , K and $N^{(l)}$, compute $\hat{k}^{\text{start}}$ and $\hat{k}^{\text{stop}}$
- 3: Retrieve the filter $\hat{w}$ based on the parity of h^*
- 4: **for** ($\hat{k} \in [\hat{k}^{\text{start}}, \hat{k}^{\text{stop}}]$) **do**
- 5: From h^* , K and $\hat{k}$, compute the output index h
- 6: $x[h] = x[h] + \hat{w}[\hat{k}]$
- 7: **end for**
- 8: **end for**

Figure 5.12: Sparse 1D convolution layer operation

5.1.3.2 Event-based 2D multi-channel convolution

Adaptation of the previous results for the case of 2D convolution is rather straightforward ; two axes now have to be considered. The division between odd and even parts occurs, only that now there exist 4 possibilities, as illustrated in Fig. 5.13. In order to obtain the channel, height and width index of the event, they must be computed from the global index that the update method in Fig. 5.4 outputs. Unfortunately, this computation involves integer

divisions between variables and are costly to execute. Section 5.2 presents a solution to this problem.

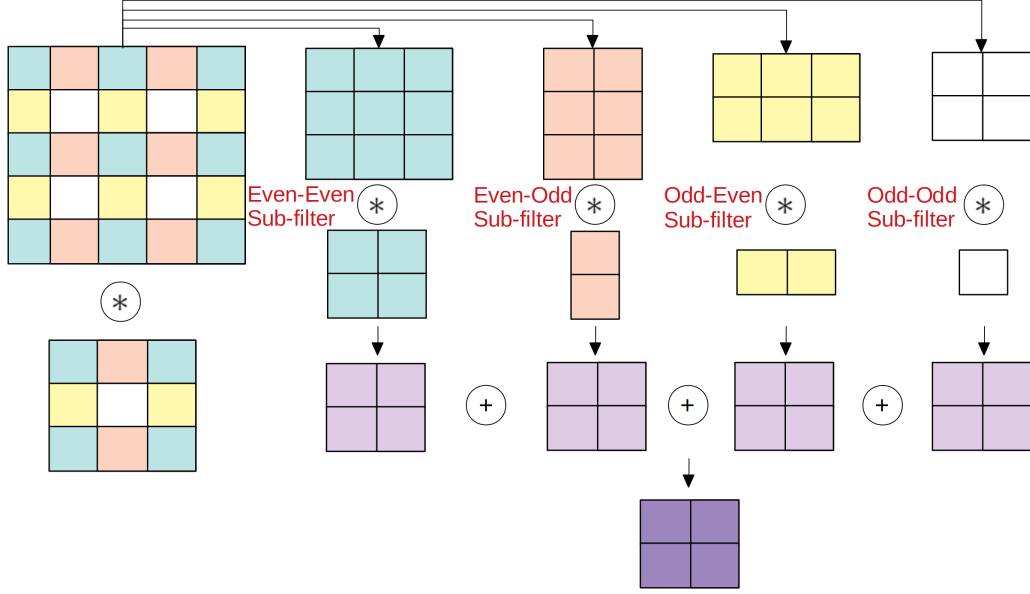


Figure 5.13: Even and odd filter separation in 2D stride 2 convolution.

Moreover, an additional loop to account for the different output channels is necessary. In order to compute the quantities of Eq. 5.34 to 5.39 only once per event, the channel loop must be the most inner loop. This allows the use of SIMD instructions in Section 5.2.

The pseudo-code of Fig. 5.14 presents the operations necessary for the 2D event-based convolution. Details are omitted since it mainly builds upon the code in Fig. 5.12.

Input: $\mathcal{E}^{(l)}$, $\mathbf{W}$, K_h , K_w , $C^{(l+1)}$, $C^{(l)}$, $N_h^{(l)}$, $N_w^{(l)}$

Output: $\mathbf{X}$

```

1: for (every event global index  $g^* \in \mathcal{E}^{(l)}$ ) do
2:   From  $g^*$ ,  $C^{(l)}$ ,  $N_h^{(l)}$  and  $N_w^{(l)}$ , compute spatial indices  $c^*$ ,  $h^*$ ,  $w^*$ 
3:   From  $(h^*, w^*)$ ,  $K_h$ ,  $K_w$ ,  $N_h^{(l)}$  and  $N_w^{(l)}$ , compute  $\hat{k}_h^{\text{start}}$  and  $\hat{k}_h^{\text{stop}}$  for  $\hat{k}_h$  and  $\hat{k}_w$ 
4:   Retrieve the filter  $\hat{W}$  based on the parity of  $h^*$  and  $w^*$ 
5:   for ( $\hat{k}_h \in [\hat{k}_h^{\text{start}}, \hat{k}_h^{\text{stop}}]$ ) do
6:     From  $h^*$ ,  $K_h$  and  $\hat{k}_h$ , compute the output index  $h$ 
7:     for ( $\hat{k}_w \in [\hat{k}_w^{\text{start}}, \hat{k}_w^{\text{stop}}]$ ) do
8:       From  $w^*$ ,  $K_w$  and  $\hat{k}_w$ , compute the output index  $w$ 
9:       for ( $c \in [0, C^{(l+1)} - 1]$ ) do
10:         $X[c, h, w] = X[c, h, w] + \hat{W}[c, c^*, \hat{k}_h, \hat{k}_w]$ 
11:       end for
12:     end for
13:   end for
14: end for

```

Figure 5.14: Sparse 2D convolution layer operation

5.1.3.3 Complexity comparison

The complexity of convolution operation of Eq. 5.19 is given by

$$\text{FLOP}_{\text{Conv. Normal}} = 2 \cdot C^{(l+1)} \cdot N_h^{(l+1)} \cdot N_w^{(l+1)} \cdot C^{(l)} \cdot K_h \cdot K_w \quad (5.40)$$

where the output width and height are given by

$$N_h^{(l+1)} = \frac{N_h^{(l)} - K_h}{2} + 1$$

$$N_w^{(l+1)} = \frac{N_w^{(l)} - K_w}{2} + 1$$

On the other hand, the complexity varies depending on the location of input events. The worst case scenario is for events of even indices (longer filter) and in the center spatial region of the input (no border effect). In this case, the complexity of the sparse algorithm is given by

$$\text{FLOP}_{\text{Conv. Sparse}} = |\mathcal{E}_n^{(l)}| \cdot C^{(l+1)} \cdot \frac{K_h + 1}{2} \cdot \frac{K_w + 1}{2} \quad (5.41)$$

where $|\mathcal{E}_n^{(l)}|$ is at most equal to the number of input neuron

$$N^{(l)} := C^{(l)} \cdot N_h^{(l)} \cdot N_w^{(l)} \quad (5.42)$$

A comparison of the complexity of both algorithms is shown in Fig. 5.15 for parameters of the first convolutional layer of the optimized model (Section 3.4.4).

- $C^{(l)} = 2$
- $N_h^{(l)} = 41$
- $C^{(l+1)} = 64$
- $N_w^{(l)} = 53$
- $K_h = K_w = 5$

It shows the superiority of the sparse method for event-based vectors. The reduction in FLOP is inversely proportional to the sparsity of the input. On the other hand, the meticulous integer computations necessary to determine the addressing in the vectors is not taken into account in this metric. Nevertheless, those indexing computations only appear once per processed event and could be negligible compared to the several floating-point operations inside the loop.

5.1.4 Target application structure

This chapter has established so far alternative methods to realize the different operations of a Slayer layer. The FLOP complexity of the convolution and dense operations have been reduced and made proportional to the input sparsity while the filtering of the spike signals have been translated into an IIR model that can operate one time-step at a time.

The last goal of this section is therefore to create a complete SNN in C++ that comprises multiple convolutional and dense layers. In order to achieve a modular yet efficient framework,

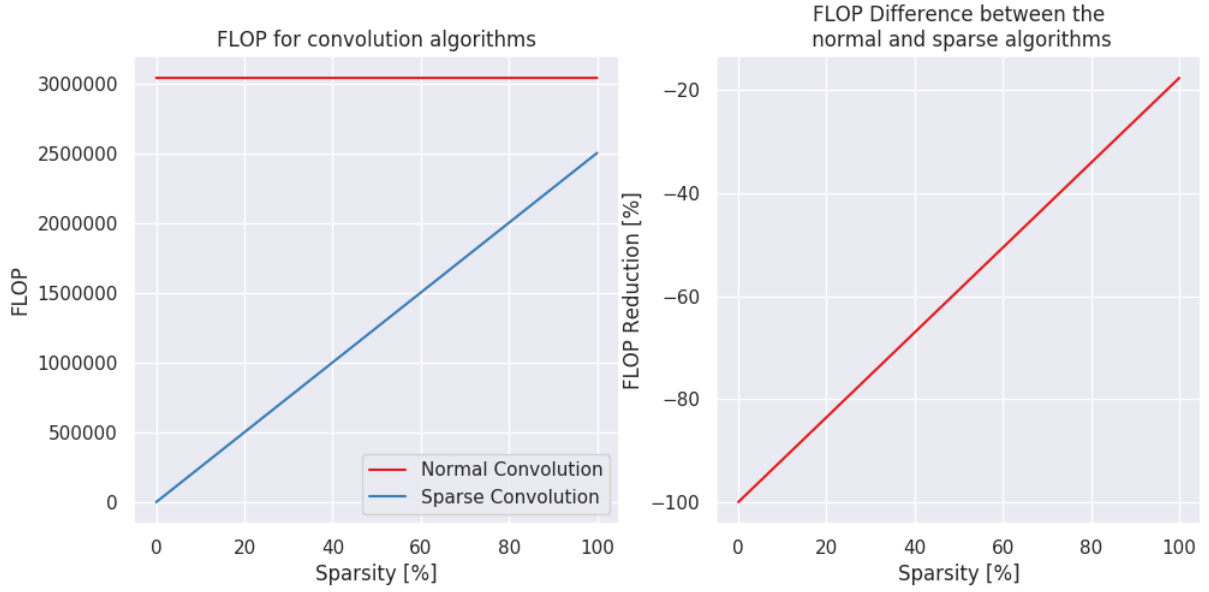


Figure 5.15: FLOP of the normal and sparse convolution algorithms.

the software created for the Cortex-A9 mixes object-oriented features of C++. Class interfaces and inheritance constructs are used to reflect the model's hierarchy, with dedicated optimized functions for the layer and update operations. The structure of the software is described in the following paragraphs with a particular focus on the interfaces.

5.1.4.1 Model and layer interface

A C++ interface called `ILayer` is written to define the variables that are shared in-between hidden layers as well as for the input and output of the complete model. An interface does not provide an implementation but rather constraints the arguments and return-types of methods that are then specifically implemented by inheritance. The fundamental elements of the layer interface are specified below :

- The **forward** method triggers all the operations associated with a single time-step. Namely :
 1. The sparse layer operation that uses the input event indices and produces the IIR filter input.
 2. The update operation that takes the filter input and updates the states of every neuron in the layer. It produces an output array of new event indices that can be used in the next layer.

The forward method therefore takes as input a `Spike_s struct` pointer and does not return any value (`void` function).

- The `Spike_s struct` is a custom structure defined in Fig. 5.16.

It contains a pointer to an array of event indices that is fixed in size and a second variable that indicates the number of events inside the array. This is therefore a read/write structure that is used to transmit the events of a single time-step between

```

1 struct Spike_s
2 {
3     unsigned int *array_spike;
4     unsigned int len_spike;
5 };

```

Figure 5.16: C++ Spike structure

layers. The output event indices from one layer are written to this structure and the next layer reads those events for its layer operation. This is therefore an implementation of the $\mathcal{E}_n^{(l)}$ set introduced in the above sections.

- The **reset** method resets the states of layer's neurons. This is necessary if the model operation is stopped and a different sample needs to be processed.

5.1.4.2 Model class

The **Model** class inherits from **ILayer** and contains an ordered list of layers. Its role is two-fold :

1. The **forward** method receives the **Spike_s struct** containing the input events of a single time-step of the DVS camera, it therefore needs to forward this structure sequentially to every layer of the model. At last this same structure therefore contains the output events of the model. Similarly, the **reset** method causes a general reset for every layer.
2. The **get_class** method uses the output events to infer which class is predicted by the model. It does so at every time-step by accumulating the number spikes of each output neuron ; the network is therefore capable of providing a prediction result at every time-step.

5.1.4.3 Layer base class

A **BaseLayer** class inherits from **ILayer** and provides the common structure to both dense and convolutional layers.

- It contains two pointers that indicate the region in memory that stores the weights of the layer and the IIR filter parameters.
- It allocates a region in memory for the following elements :
 1. An **input** array of length $N^{(l+1)}$ that filled at every time-step by the sparse layer operation.
 2. A **state** array of length $4 \cdot N^{(l+1)}$ that contains the states of the IIR filter for every neuron of the layer. There are two states per filter (Fig. 5.3) and two filters per neuron (Fig. 5.1), thus 4 entries per neuron.

- It contains a pointer to a specialized `layer_operation` function that implements either the dense or convolution operation.
- It provides an implementation for the `reset` method that resets the entries of the `input` and `state` arrays.
- It provides an implementation for the `forward` method that is pretty straightforward, detailed in Fig. 5.17.

```

1 void BaseLayer::forward(Spike_s *spike)
2 {
3     // spike (Spike_s*) : pointer to the spike structure containing events
4     // input  (float*) : pointer to the array of iir filter inputs
5     // weight (float*) : pointer to the array of weights
6     // state  (float*) : pointer to the array of iir filter states
7     // filter (float*) : pointer to the array of iir coefficients
8     // len_layer (uint) : number of neurons in the layer
9
10    // dense or conv operation : writes into input
11    (*layer_operation)(input, weight, spike);
12
13    // update operation : resets input, writes into states and array_spike
14    spike->len_spike = update_state(input, state, filter, spike->array_spike, len_layer);
15 }

```

Figure 5.17: C++ forward layer operation

The read/write behavior of the `spike` struct is readily visible in Fig. 5.17. The same memory region is used for every layer and limits memory usage. The `input` array is filled by the layer operation and used by the update method. However, it must be cleared before it being used for the next time-step because, as explained in Section 5.1.3.1, the entries are accumulated during the processing of the input events.

5.1.4.4 Specialized layer

The `DenseLayer` and `ConvLayer` classes inherits from `BaseLayer` and provides the following arguments for its initialization :

- The pointer to the region in memory that stores the weights of the layer.
- The number of neurons $N^{(l+1)}$ necessary to create the `input` and `state` arrays.
- The adequate `layer_operation` method.

The array of filter parameters is directly specified to the `BaseLayer` class as a `static` member since the filter parameters are common to every layer.

5.1.4.5 Weights, input samples and predictions

The `Model` class is instantiated by the main method of the Baremetal Framework established in Section 4.6. After the initialization sequence of the cache, MMU and SD/MMC, the filter

and weights are copied from the FAT partition of the SD/MMC into the SDRAM. The model is then instantiated with the parameters of the optimized model found in Section 3.4.4 and the `spike_struct` is allocated with an array size equal to the number of neurons found in the largest layer.

Assuming at this point that the model uses single precision floating-point values for its weights and operations, and the event indices are recorded in 32-bit integers, the model instantiation therefore has the following memory footprint detailed in Table 5.1. In addition, the memory repartition in percentage is illustrated in Fig. 5.18.

Category	Conv. 1	Conv. 2	Dense 1	Dense 2	Spike Struct	Total
Weight [#]	3200	102400	720896	256		826752
Weight [MB]	0.0128	0.4096	2.8836	0.0010		3.3070
Input & States [#]	152000	28160	640	10		180810
Input & States [MB]	0.6080	0.1126	0.0026	0		0.7232
Spike Struct [#]					30400	30400
Spike Struct [MB]					0.1216	0.1216
Total [MB]	0.6208	0.5222	2.8861	0.0011	0.1216	4.1518

Table 5.1: Memory repartition for the optimized model.

The program therefore uses at minimum 4.1518 MB of heap space, a major part of which is occupied by the model weights. If it were not for the first dense layer, this trend would be the opposite as the weights of this layer occupies 2.8836 MB. This result also indicates without surprise that the memory in the heap cannot reside entirely inside the L1 and L2 caches that have a capacity of 32 + 512 KB. Moreover, the L2 cache shares its cache space with the program’s instructions. A careful planning of the memory usage is therefore necessary to take profit of the L1 and L2 caches ; it is the subject of Section 5.2.

After initialization of the model and the `Spike_s_struct`, the program calls the `forward` for every time-step. The sample events for a given time-step are fetched from the FAT partition before being processed by the network. In this configuration the model accounts for the fact that the events should in practice be retrieved just before its processing from a volatile memory. Since this work did not focus on implementing a direct interface with a DVS camera, possibly using the FPGA part of the Cyclone V SoC, this structure remains close to what an actual embedded application would perform. After the last time-step of the sample, the model is reset and process the next one. Thanks to the 32 MB space of the FAT16 partition, some hundreds of samples can exploited to derive sufficient statistics about the model. The next section focuses on the timing performances of the model and proposes optimization solutions using the NEON instruction set of the Cortex-A9 processor.

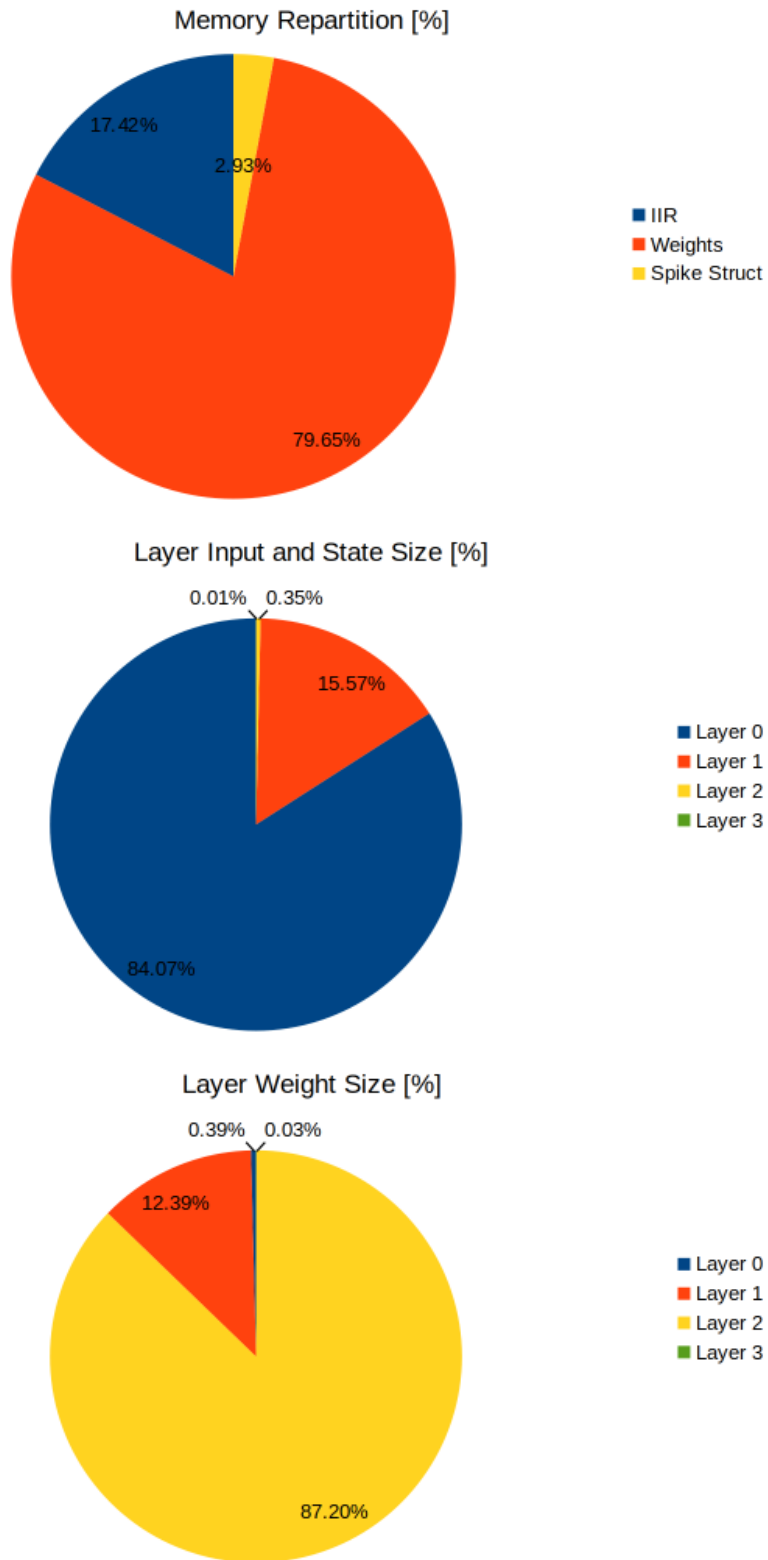


Figure 5.18: Memory repartition in percentage for the optimized model, divided for layers and operations. Absolute values are in Table 5.1.

5.2 NEON Optimization

A low-complexity implementation of the Slayer algorithm has been established in the last section ; this solution needs to be profiled in order to derive performance metrics and seek possible optimizations. Prior to any measurement, several modifications related to the arrangement of the **state** and **input** data in memory are performed in order to obtain a more efficient cache usage and avoid unnecessary operations.

5.2.1 Prior optimizations

5.2.1.1 Update method

Going back to the C++ snippet in Fig. 5.4, two important observations are made

1. The method takes numerous arguments, the major part of which is related to the state pointers and the filter coefficients. Section 4.2.4 stated that ARM's ABI allows to use up to four 32-bit registers to pass arguments to a function. The remaining arguments must be loaded on the stack. These data transfers may therefore be avoided by regrouping arguments into arrays.
2. After the IIR update equations, the states of the filter are shifted in the loop. These operations happen for every neuron in the layer and exert a strong impact on performance. They can be easily regrouped with a smart memory organization.

In order to address these points, the following solutions are proposed :

- A common **filter** array of size 7 regroups the coefficients of filters ϵ and ν as well as the threshold θ .
- A common **state** array is used in the argument list. This array contains 3 pointers to the memory region allocated for the layer.
 1. **state**[0] is the filter input and points to a region of size $N^{(l+1)}$.
 2. **state**[1] regroups the first IIR states (**s_1**) of the ϵ and ν filters and points to a region of size $2 \cdot N^{(l+1)}$.
 3. **state**[2] regroups the second IIR states (**s_2**) of the ϵ and ν filters and points to a region of size $2 \cdot N^{(l+1)}$.

The ϵ and ν states are interleaved in memory, optimizing the cache line usage since they are accessed together at every iteration of the loop. The layer memory can therefore be a contiguous block of size $5 \cdot N^{(l+1)}$. This arrangement is illustrated in Fig. 5.19.

- Since the memories of IIR filters are now regrouped by delay (**s1-s2** separation), every state may be shifted at once by swapping the pointer heads in the **state** array at the end of the function. This is shown with the red arrow in Fig. 5.19.

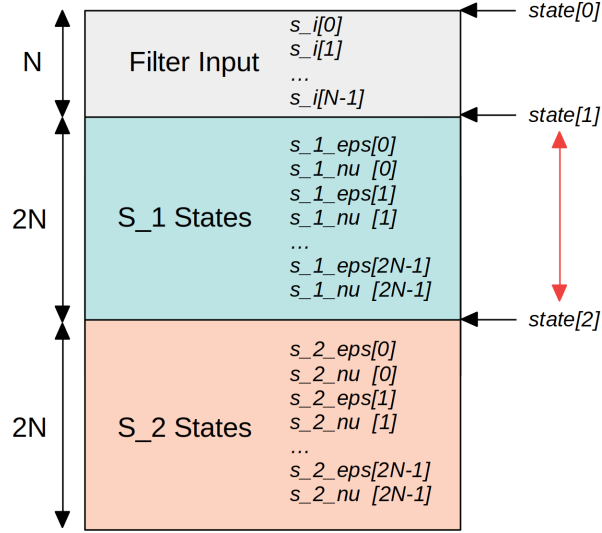


Figure 5.19: Layer's **input** and **state** memory organization. **state**[1] and **state**[2] permute at the end of the update method.

The enhanced solution is presented in the C++ snippet of Fig. 5.20 ; values previously referred as **s0** are directly written in **s2** and the pointer heads are swapped at the end of the loop. In addition, we can see that it is the update operation's responsibility to clear the entries of the **input** array after use. Since the address is already cached at this point, it is believed to provide good performances.

5.2.1.2 Convolution operation method

Section 5.1.3 established the sparse convolutional algorithm and highlighted the following specificity. When the **update_state** method registers output events in the **Spike_s struct**, it does so by storing the neuron's index of the corresponding **input** array of the layer. However, when processing the event, the convolutional operation requires the indices of the channel, height and width dimensions in order to access the correct filter and output memory locations. This necessitates a conversion of the *flattened* index to *spatial* indices, involving integer division and multiplies. The division operation is not associated with any instruction on the Cortex-A9 since there is no hardware dedicated to this operation. As a result, the division is carried by a call to a subroutine of the GCC library, taking several tens of cycles [88] to execute. This problem is avoided if the divisor is known at the compile time. The GCC compiler can replace the division with a combination of **MUL** and **LSR** instructions. This implies that the **layer_operation** method must be specialized for the dimensions of a unique layer. This is done using the template meta-programming ability of the C++ language. A dedicated function for **conv_operation** is created for each convolutional layer in the model. This inevitably results in a larger executable file size and could possibly worsen the efficiency of the L1 Instruction cache. This solution is maintained and later discussed when measuring the performances of the method.

Secondly, Eq. 5.19 insists on the fact that the weight tensor in the multi-channel convolution is 4-dimensional. Since an array in memory is always 1-dimensional due to the flat memory addressing, the organization of this kind of multi-dimensional structure is important in order to offer high performances. A multi-dimensional array is *flattened* when stored in memory

```

1 unsigned int update_state(float** state, const float* filter, unsigned int* output, const↵
    unsigned int len)
2 {
3     // Number of output spikes
4     unsigned int len_output = 0u;
5
6     float* s1{state[0]};
7     float* s2{state[1]}; // eps and nu states are interleaved
8     float* s2{state[2]};
9     unsigned int n;
10    for(n = 0;
11        n < len;
12        ++n, s1 += 1, s2 += 2, s2 += 2)
13    {
14        bool spike = ((filter[0] * s1[0] + filter[1] * s1[1]) >= filter[6]);
15        if (spike)
16            output[len_output++] = n;
17
18        s2[0] = s1[0] + filter[2] * s1[0] + filter[3] * s2[0]; // eps iir
19        s2[1] = ((float) spike) + filter[4] * s1[1] + filter[5] * s2[1]; // nu iir
20        s1[0] = 0.0f;
21    }
22
23    // Update states s2 <-> s1 (change pointers head)
24    float* temp = state[1];
25    state[1] = state[2];
26    state[2] = temp;
27
28    return len_output;
29 }

```

Figure 5.20: Prior optimization for the IIR update method.

and the mapping respects the following rule : only the last dimension is contiguous in memory while the higher dimensions are interleaved. This is illustrated in Fig. 5.21 for a 2D array.

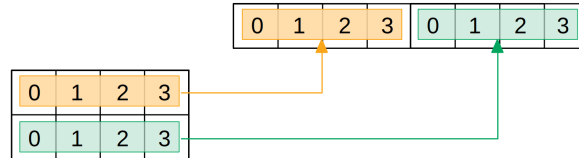


Figure 5.21: Indexing of a 2D array in memory ; the values arranged in the width dimension are contiguous.

In the case of the 4D weight tensor of the convolution operation, we can deduce the ideal ordering of axes based on the following observations :

- The event's channel index determines which 3D-slice of the 4D weight tensor is added to the feature map of the layer.
- The event's height and width indices determine which sub-filter is used, the indices are then iterated through in ranges dictated by the filter size and border conditions.
- The output channel index is not determined by the events and all indices of this dimension are iterated through.

The most logical axis ordering is therefore the following :

- The first dimension must be the input channel index.
- Then the 3D-tensors of the sub-filters are organized based on the parity tests of the event's height and width indices. This could be viewed as an intermediate *sub-filter dimension*.
- The third and fourth dimensions are the sub-filter's height and width. The relative ordering of the two irrelevant. The height is selected before the width in order to remain consistent with the indexing previously introduced and also present in the Pytorch Framework.
- The last dimension is for the output channel. This choice is fundamental since :
 1. The indexing of this dimension does not depend on the input event. Looping in this dimension is therefore deterministic and can be parallelized.
 2. The size of this dimension can be easily controlled by the model's parameters. The channel size can be selected to be a multiple of 4 in order to leverage SIMD instructions as seen in Section 5.2.6.2.

The axis ordering of the weights is illustrated in Fig. 5.22 for a filter with a spatial size of 5×5 . Similarly, the feature map array shares the same axis ordering. The innermost loop of the previously presented algorithm of Fig. 5.14 is therefore the following :

$$X[h, w, c] = X[h, w, c] + W[c^*, p, \hat{k}_h, \hat{k}_w, c] \quad (5.43)$$

where p is defined by the parity test on h^* and w^* .

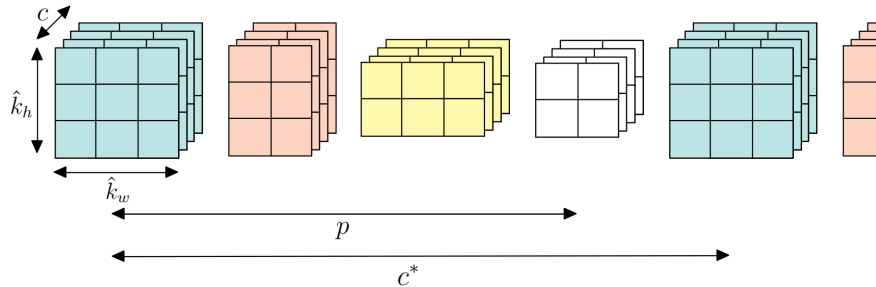


Figure 5.22: Weight tensor axis ordering for a 5×5 filter.

The weights of the optimized model in the Pytorch framework must be converted into this format, which is easily done with an interface Python script. There is however one last caveat to consider that is not the least : the Pytorch convolutional layer operation is really in fact a cross-correlation operation. Therefore, when preparing the weight tensors for this algorithm, spatial dimensions must be inverted.

5.2.2 Profiling of the initial solution

The program is now ready for its first profiling in order to identify the bottleneck in timing performances. A quick recall of the compiler flags and memory configuration used in the Bare-metal framework is given in Table 4.11. As the implementation used at this point is

entirely written with C/C++ standard libraries, it will be named the *C implementation* or *C algorithm* for the rest of these sections.

The Barman library is used to annotate the different sections in the program and provide accurate timing measurements for each method, using the 200 MHz MPU peripheral clock of the Cortex-A9. The measurements therefore have a 5 nanoseconds accuracy but inevitably include a small timing overhead caused by the sampling function. The profiling is realized on 36 samples from the test-set of N-CARS due to the limited memory buffer allocated to that effect. The timing results are displayed in Fig. 5.23 to 5.27.

This analysis reveals several important facts :

- The execution time of the **update** method is larger than the one of the **layer_operation** for every layer (Fig. 5.23). It accounts for 84.9 % of the average execution time (Fig. 5.26).
- Since the computational complexity of the **update** method is only related to the layer's size, the convolutional layers, comprising the highest number of neurons take 89.1 % of the average execution time (Fig. 5.25).
- The execution time of the **update** method has a standard deviation of at most 300 μ s (Fig. 5.23). This method should have a consistent execution time but this behavior is due to the pseudo-random replacement policy of the caches.
- Although the first dense layer has the largest weight array size, its **layer_operation** method execution time remains by one order of magnitude lower than the **update** method of the convolutional layers (Fig. 5.27).
- The average execution time of the complete **forward** method is 4200 μ s ; this is two times larger than the sampling period of the model (Fig. 5.24).

The objective of this work is to realize a real-time solution for the Slayer model, where each forward pass takes an execution time strictly lower than the sampling period of the model, which is 2000 μ s. Fortunately, it remains in the same order of magnitude than the timing performance of the current solution. Therefore, our resources are focused on optimizing the **update** method that is responsible for the major part of the current execution time. Satisfactorily, it is also the method for which the complexity is a function of the layer's size ; the memory accesses are sequential, the loop sizes are deterministic and therefore a SIMD optimization should offer best performances.

As a second step, the **layer_operation** methods can be improved, with a particular focus on the convolution operation.

5.2.3 Update method optimization for 32-bit floating-point

5.2.3.1 Vectorized compiler approach

The first approach to seek better performances for the **update** method is to rely on the GCC compiler and its automatic vectorization tools to produce SIMD instructions. Auto-

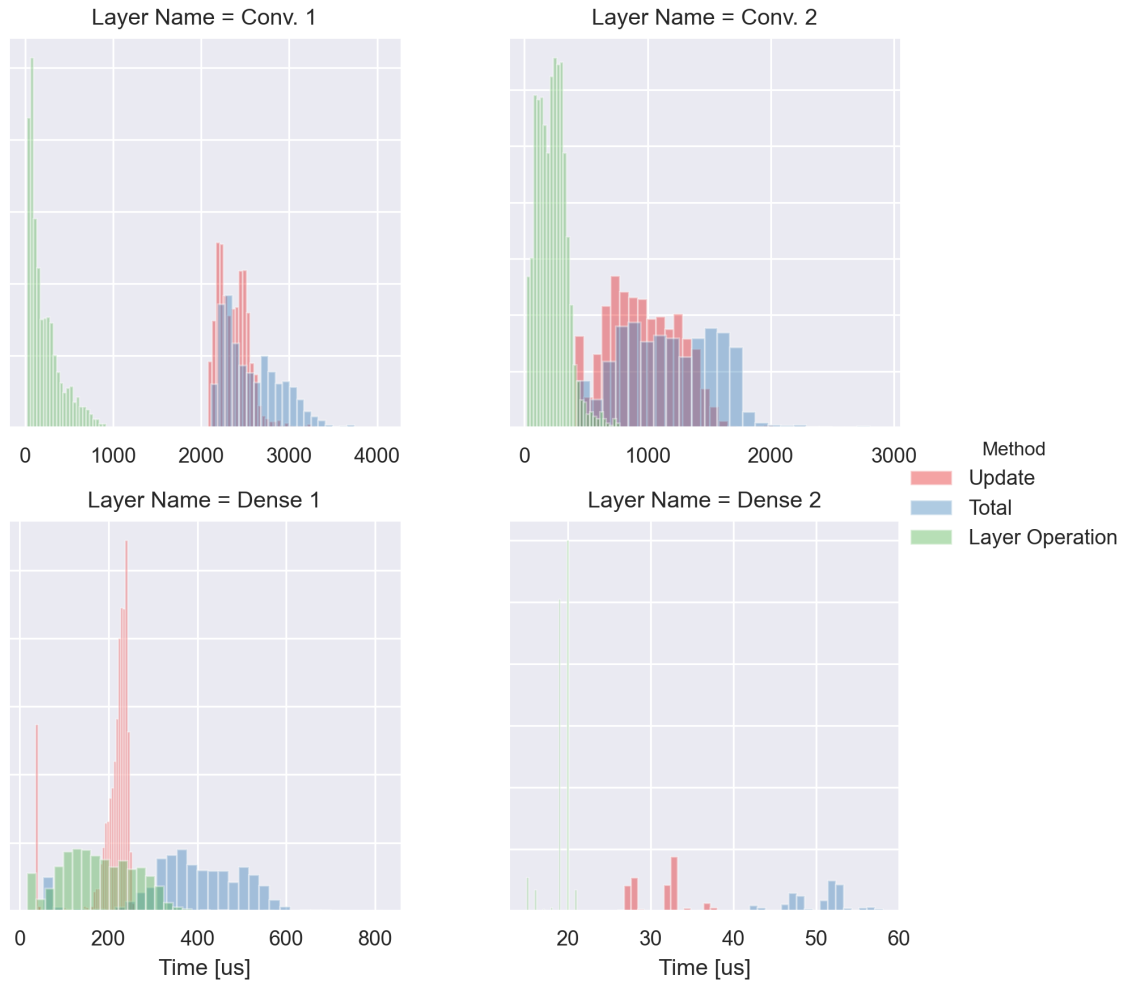


Figure 5.23: Execution time distribution per layer for the C implementation.

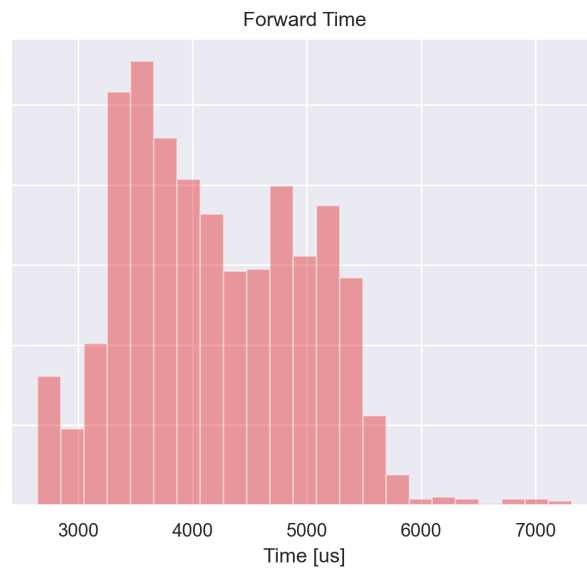


Figure 5.24: Execution time distribution of the **forward** method for the C implementation.

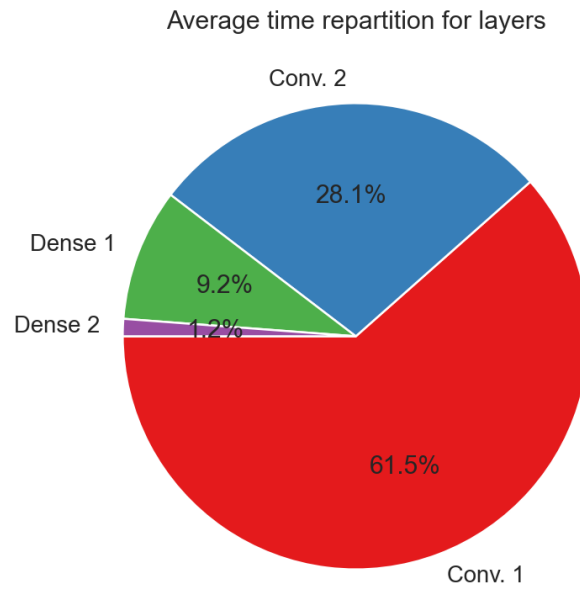


Figure 5.25: Average execution time repartition for the different layers for the C implementation.

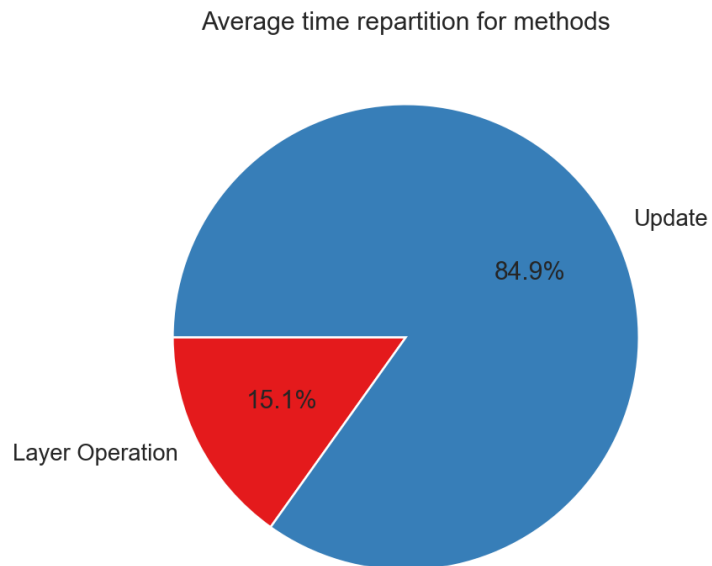


Figure 5.26: Average execution time repartition for the `layer_operation` and `update` methods for the C implementation.

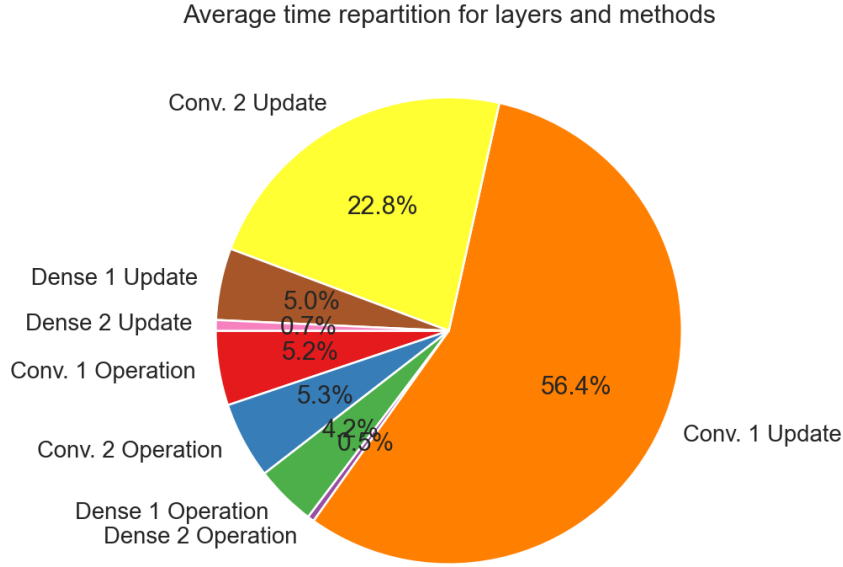


Figure 5.27: Average execution time repartition for the different layers and methods for the C implementation.

vectorization is triggered by the `-ftree-vectorize` flag and the additional `-f-unsafe-math-optimization` in order to produce floating-point NEON SIMD instructions. Both of those flags are in turn included under the `-Ofast` flag (Section 4.6.4). The initial solution of Section 5.2.2 is already compiled with the corresponding optimization flag but a careful analysis of the produced executable should be conducted in order to verify that it produces the intended instructions.

The inspection of the machine language generated for the `update` method reveals that the compiler produces VFP floating-point instructions that uses 32-bit `S` registers. Instead, the compiler should theoretically leverage the use of 128-bit `Q` registers that offer operations on 4 floating-point elements at the same time. The vectorization compiler produces the best results when it can deduce the intentions of the programmer ; Section 4.3.5 provides a set of rules that should be adopted in order to maximize the chance of having a code vectorized by the compiler. However, even with a careful planning of the C++ code, a constant failure of the compiler to produce SIMD instructions was observed. The produced assembly instructions is shown in Appendix C.1.1.1. This failure was traced back to the following causes :

1. The `if` clause responsible for comparing the membrane potential with the threshold and for storing the output event's indices cannot be vectorized. The compiler uses by default the `VCMPE.F32` and `VMRS` instructions in order to compare two floating-point numbers and transfer the `N,Z,C,V` flags back to the ARM pipeline where the event's index is stored.
2. The compiler seems to be confused by the large number of C++ statements present inside the loop. Further testing has shown that a correct vectorization appears for

loops that contain a very restricted number of statements.

The approach that relies on the GCC vectorizer is not conclusive and did not provide an increase in performance. Another strategy is adopted in the next section in order to produce reliable SIMD instructions.

5.2.3.2 NEON Intrinsics approach

In order to adapt the update algorithm using NEON intrinsics, a few preliminary computations must be carried regarding the number of available registers. As explained in Section 4.3.3, the NEON unit comprises thirty-two 64-bit D registers that can also be interpreted as sixteen 128-bit Q registers. Knowing the limited amount of SIMD registers, the `update` method must be analysed beforehand to reveal whether the required variables are able to fit inside the registers. Assuming the algorithm uses the 32-bit floating-point data-type, it can therefore be parallelized with at most 4 lanes processed in parallel in Q registers. Starting from the implementation of Fig. 5.20, the following analysis counts the number of D and Q registers necessary :

- 3 D registers for the 6 filter coefficients.
- 1 Q register for the `input` array.
- 1 Q register for the `spike` array.
- 4 Q registers for the filter states `s1` and `s2`.

Using about 9 Q registers out of the 16 available, a 4-lane parallelization leaves 7 additional registers for intermediate variables. With this repartition, the compiler can easily allocate a register with each intrinsic data-type, without the need of pushing registers on the stack. In fact, stack operations inside the loop could turn out to be very costly if the memory addresses are not kept in the cache. Stack operations should therefore be as limited as possible. The following paragraphs skim through the constitutive elements of the proposed implementation with NEON intrinsics. The complete code associated with this description is available in Appendix C.1.1.2.

Loop invariants At the beginning of the method, the filter coefficients must be loaded into NEON registers for their later use inside the loop. Similarly, the threshold θ as well as the constants 0.0 and 1.0 must be loaded in order to be used for the comparison with the membrane potential. These actions are illustrated in the C++ snippet in Fig. 5.28, annotations indicate the element's structure inside the registers.

Load from memory At the beginning of every loop, a block of the filter input and state arrays must be loaded into Q registers. As detailed in Section 4.3.4, the VLD instruction can load up to 2 Q registers at the same time if the interleaving modifier is used. Therefore, if the ϵ and ν states are interleaved in memory, they can be de-interleaved with a single VLD2.32 instruction. This process is illustrated in Fig. 5.29 for the states of `s1`.

```

1 // 63      32 31      0 : D register
2 float32x2_t vforw_duo = vld1_f32(filter); // [forw1_nu forw1_eps]
3
4 float32x2_t vback_eps = vld1_f32(filter + 2); // [back2_eps back1_eps]
5 float32x2_t vback_nu  = vld1_f32(filter + 4); // [back2_nu  back1_nu  ]
6
7 // 127 96 95 64 63 32 31 0 : Q register
8 float32x4_t vtheta = vld1q_dup_f32(filter + 6); // [theta theta theta theta]
9
10 // Spike values
11 float32x4_t vones  = vmovq_n_f32(1.0f); // [1.0f 1.0f 1.0f 1.0f ]
12 float32x4_t vzeros = vmovq_n_f32(0.0f); // [0.0f 0.0f 0.0f 0.0f ]

```

Figure 5.28: Loop invariants initialization.

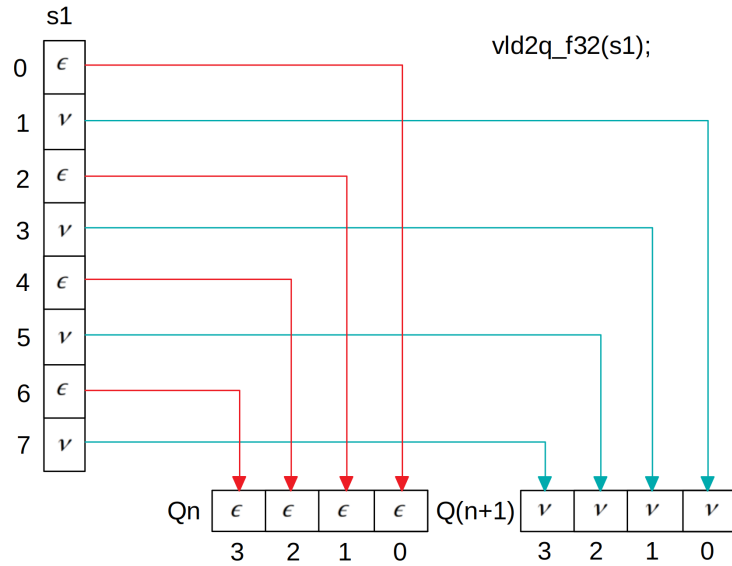


Figure 5.29: Memory de-interleave of states $s1$ into 2 Q registers.

The same process is applied to **s2**. The further advantage is that 8 floating-point elements that are fetched have a width equal to the L1 cache line. This has a great potential for cache pre-fetching and will be exploited below. The filter input is not interleaved and 4 entries are loaded using a **VLD1.32** instruction. The input is immediately cleared after the load for the subsequent time-steps. These instructions are shown in the snippet of Fig. 5.30.

```

1 float32x4x2_t vs1 = vld2q_f32(s1); // Deinterleave (eps | nu | eps | nu ...)
2 float32x4x2_t vs2 = vld2q_f32(s2);
3 float32x4_t vsi = vld1q_f32(si);
4 vst1q_f32(si, vzeros); // Clear input while still cached

```

Figure 5.30: Loading of the states and filter input into Q registers.

Output spike The next step involves computing the membrane potential and comparing it to the threshold to produce output events, also referred to as output spikes. The indices of spiking neurons are stored inside the **output** array and are used in subsequent layers. The membrane potential is calculated using the **vmulq_lane_f32** and **vmlaq_lane_f32** intrinsic functions. They perform a multiply and multiply-accumulate operation using a single lane for the multiplier. Since the **FPSCR** status flags cannot be used in the SIMD configuration, the compare operation with the threshold must be carried using the **vcgeq_f32** intrinsic. The output register's lanes are either filled with ones or zeros depending on the result of the comparison. This bit-mask is then used to build the input of the ν filter. The **vbslq_f32** intrinsic selects either the lanes of registers **vones** or **vzeros** depending on the bit-mask.

At last, the bit-mask is stored in a buffer array on the stack with the **vst1q_u32** intrinsic. A small inner loop is used in order to fill the **output** array of events. This process is illustrated in the snippet of Fig. 5.31.

```

1 // Forward filter equation : u = forw1_eps * s1_eps + forw1_nu * s1_nu
2 // spike = (u >= theta)
3
4 // temp = f_1,eps * s_1,eps
5 float32x4_t vspike = vmulq_lane_f32(vs1.val[0], vforw_duo, 0);
6
7 // u = temp + f_1,nu * s_1,nu
8 vspike = vmlaq_lane_f32(vspike, vs1.val[1], vforw_duo, 1);
9
10 // mask = (u >= theta) (all 1 if true, all 0 otherwise)
11 uint32x4_t vcond = vcgeq_f32(vspike, vtheta);
12
13 // spike = 1.0 if mask all 1, 0.0 otherwise
14 vspike = vbslq_f32(vcond, vones, vzeros);
15
16 // Output spikes store
17 vst1q_u32(temp_output, vcond);
18 for (unsigned int m = 0; m < 4; ++m)
19     if (temp_output[m])
20         output[len_output++] = (n + m);

```

Figure 5.31: Membrane potential computation and thresholding.

State update The last part of the loop consists in computing the new filter state for both filters. This is easily done using the **vmlaq_lane_f32** intrinsic again. The computation for

ϵ and ν filters are interleaved in order to avoid dependency stalls. Normally, the compiler would re-schedule the code accordingly if this situation occurs but it does not cost much to take it into account at this point. The new state is then written with interleaving at the address of `s2` using the `vst2q_f32` intrinsic. As previously explained back in Fig. 5.20, the pointer heads are then swapped at the end of the `update` method to account for the change of states. These steps are shown in the snippet of Fig. 5.32.

```

1 // Backward filter equation : s0_eps = input + back1_eps * s1_eps + back2_eps * s2_eps
2 //                               s0_nu  = spike + back1_nu  * s1_nu  + back2_nu  * s2_nu
3 float32x4x2_t vs0;
4
5 // Eps filter equation : temp_eps = input + back1_eps * s1_eps
6 vs0.val[0] = vmlaq_lane_f32(vsi, vs1.val[0], vback_eps, 0);
7
8 // Nu filter equation : temp_nu  = spike + back1_nu  * s1_nu
9 vs0.val[1] = vmlaq_lane_f32(vspike, vs1.val[1], vback_nu, 0);
10
11 // Eps filter equation : s0_eps  = temp_eps + back2_eps * s2_eps
12 vs0.val[0] = vmlaq_lane_f32(vs0.val[0], vs2.val[0], vback_eps, 1);
13
14 // Nu filter equation : s0_nu   = temp_nu  + back2_nu  * s2_nu
15 vs0.val[1] = vmlaq_lane_f32(vs0.val[1], vs2.val[1], vback_nu, 1);
16
17 // Store result in the s2 state memory
18 vst2q_f32(s2, vs0); // Interleave (eps | nu | eps | nu ...)
```

Figure 5.32: States update and storage.

5.2.3.3 Inspection of the Assembly code

As in the vectorizer approach, a careful examination of the produced Assembly code is necessary to ascertain that the correct instructions are inferred by the compiler and that the variables are correctly spread across the available registers. This analysis necessitates several remarks :

- The intended instructions are inferred by the compiler ; the algorithm therefore works with SIMD instructions, offering a 4-lane parallelization.
- The inner loop dedicated at filling the `output` array of events is unrolled by the compiler to avoid branch miss-predictions and to allow pre-computations of indices. The `STR` instructions are appended with conditional execution flags to implement the `if` clause.
- The `VLD` and `VST` instructions are re-scheduled in-between the data processing instructions for an optimal use of the dual-issue pipeline.
- Data processing instructions are re-scheduled in-between ARM instructions belonging to the inner loop in order to avoid dependency stalls.
- ARM registers R0 to R15 are used in order to perform pre-computation of addresses. Because of the ARM's ABI, the register's content needs to be pushed on the stack at the beginning of the function and popped at the end. This introduces a small overhead that is compensated by the gain these registers offer inside the loop.

- When counting the number of registers that are expected to be used with the intrinsics, 27 D registers should be used at maximum. However a few `VMOV` instructions are spotted in the Assembly code, meaning that some registers are re-used. Although these instructions are not desired, they might be leveraged by the compiler to avoid dependency stalls. A re-scheduling by hand could be attempted to remove those instructions but would most probably lead to additional pipeline stalls.
- Two D registers are pushed on the stack at the beginning of the function. This is due to the ARM's ABI ; it dictates that registers `D8-15` are *callee-saved*. These registers should therefore be used last. A minimal number of registers can be used if the pair of variables `vspike-vcond` and `vs0-vs1` are merged in the snippets presented above. The code could therefore only use 21 D registers. This leaves 11 unused registers and therefore `D8-D15` could be avoided. Again, this is only achievable by writing the algorithm directly in Assembly language.
- An additional unnecessary branch instruction is located before the loop to account for the case where the loop count is zero. As for the last remark, these situations only happen once per function call and therefore have a minimal impact if the number of iterations is large.

5.2.4 NEON Intrinsics performances and enhancements

This section provides a complete test-bench of the `update` method in order to derive performance metrics and possibly seek additional performances. The principal metric that we care about at this point is the time the `update` method takes per neuron. This measure can help adapt the design of the Slayer model to ensure that the real-time constraints are satisfied.

The test-bench consists in profiling the method for layer sizes ranging from 128 to 40000 with increments of 4 neurons. This range is selected to remain within the order of magnitude of the currently selected model (see Table 5.1). The test-bench measures the execution time of the method as well as additional metrics provided by the PMU in order to obtain a complete diagnostic of the memory usage and possible bottlenecks. Results are averaged over 10 runs and each run performs a complete purge of the cache system. Finally, the method uses filter input and states data that are filled with zeros ; this will not produce any output events ; their effects are measured separately at the end of this section.

The following paragraphs provide 3 important tests :

1. The impact of the cache system and GCC optimization flags are measured in a first test. It underlines the importance of configuring the cache and optimizer flags correctly, justifying the work achieved in Section 4.6.4. Different compiler options for the same software will produce different performances. The program tested here is the C implementation.
2. The NEON intrinsic implementation is compared with the C implementation ; it shows the improvements offered by SIMD instructions.

3. Based on the analysis of the metrics in the second test, a different memory arrangement is tested. The new memory layout allows to include pre-fetch hints to overcome cache misses and boost performances.
4. The impact of the number of output events is measured for the selected method.

5.2.4.1 Cache system and optimization flags

This first test aims at comparing 3 configurations :

1. The L1 and L2 cache systems are disabled, all data transfers happen exclusively between the processor and the main SDRAM memory.
2. Caches are enabled but the compiler optimization flags are set to their default (`-O0`).
3. Caches are enabled and the compiler optimization flags are parametrized for the maximum time performance (`-Ofast`).

The results are synthesized in Fig. 5.33. The x-axis always represents the number of neurons in the layer and its ticks are marked at the bottom of the figure. The y-axis represents the metric specified in the title ; the same layout will be used again in further tests. Some metrics require additional explanations before their analysis :

- *L1 D-Cache Dep. Stalls* counts the number of cycles where the processor has some instructions it cannot issue to any pipeline, while the Load/Store pipeline has at least one pending linefill request, and no pending TLB requests [91].
- Metrics that are appended with "*per iter.*" means that the corresponding metric is divided by the number of loop iterations in the `update` method. The number of iterations corresponds to the layer size for the C implementation and the layer size is divided by 4 for the NEON intrinsics implementation.

This experiment brings the following comments :

- The execution time of the `update` method is linear with respect to the layer size, as predicted back in the FLOP analysis in Eq. 5.13.
- The cache system has an overwhelming impact on the method's performance, with a factor of at least 30 compared to the other methods.
- The `-Ofast` optimization flag main influence is to reduce the memory accesses by a factor between 2 and 3. This is achieved with a careful planning of the registers used in the loop since the unoptimized software systematically allocates a slot on the stack to each intermediate variable. The number of misses is nearly identical, they mainly correspond to the compulsory misses of the algorithm.
- L1 data evictions start when the number of neurons exceeds 1639. As the total memory usage of the layer is about 5 times the layer size (Table 5.1), this corresponds to a total size of 32 kB and correlates with the size of the L1 Data Cache.

- Similarly the L2 data evictions start when exceeding 26215 neurons. This correlates with the capacity of the L2 cache which is 512 kB.
- A few spikes in cache misses are observable ; this phenomenon has to do with cache trashing and is explained in the next section.

Considering a layer size around 30000, the execution time is around $2400\ \mu\text{s}$, which corresponds to what we previously observed in Section 5.2.2.

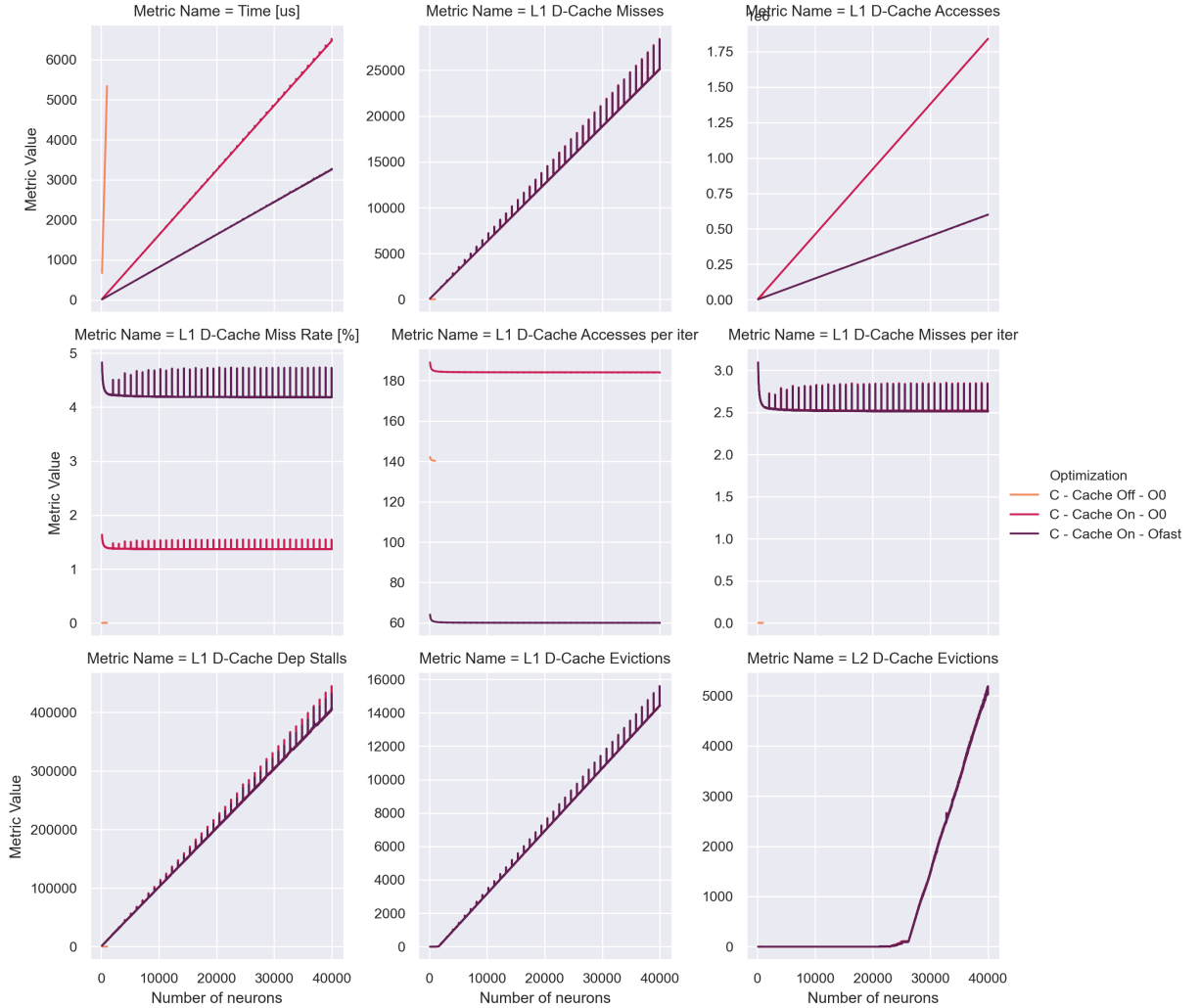


Figure 5.33: Update method metrics for the C implementation and different cache configurations and optimization flags.

5.2.4.2 NEON intrinsics implementation

The second test compares two configurations : the NEON implementation and the fastest compiled version of the C implementation. The results are shown in Fig. 5.34. This experiment provides very promising results as the NEON intrinsic implementation performs twice as fast as the C implementation. Several interesting behaviors can be observed in the other metrics :

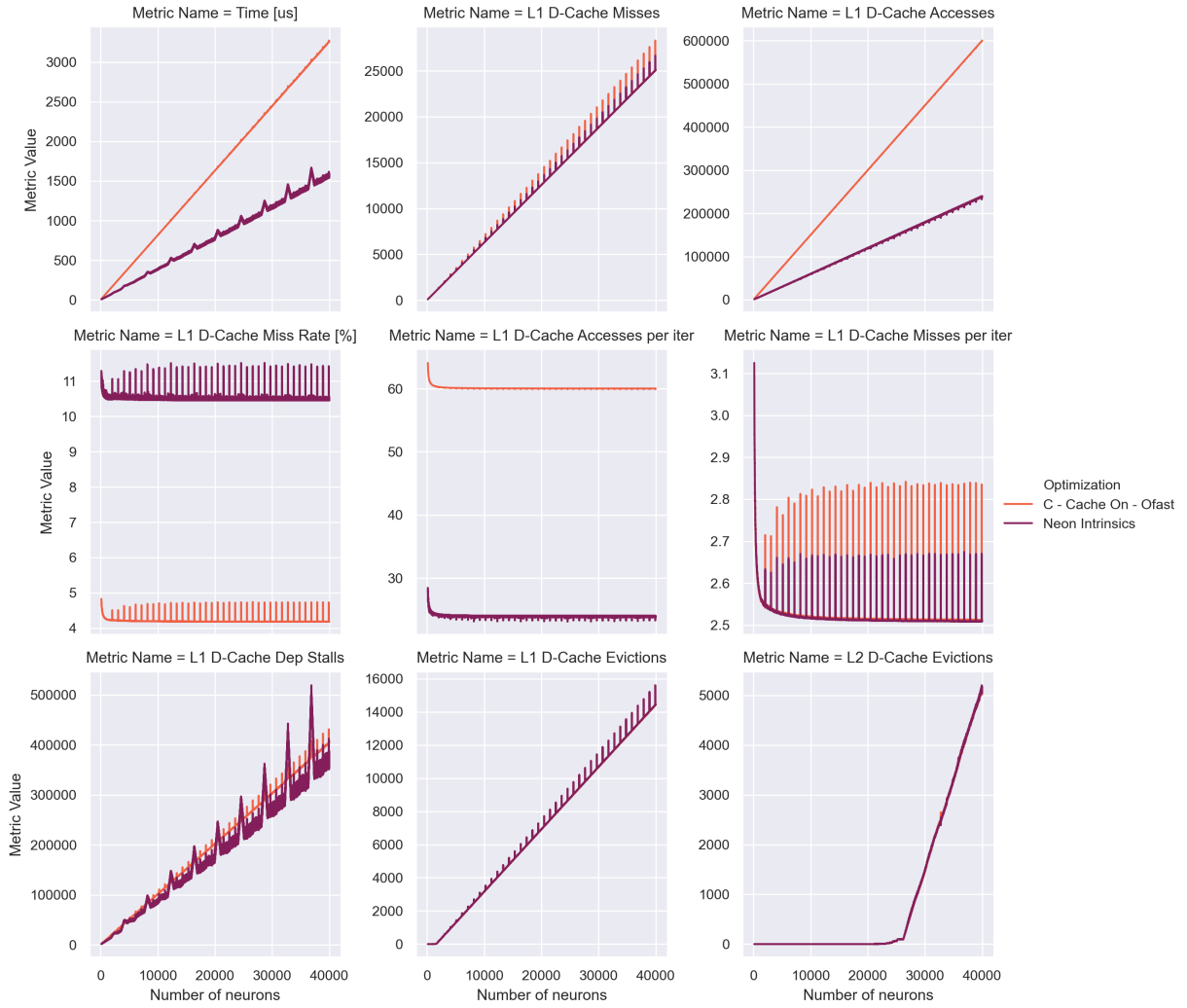


Figure 5.34: `update` method metrics comparison between the C implementation and the NEON intrinsics implementation.

- The NEON SIMD instructions allow to reduce the number of data accesses by a factor between 2 and 3. The total number of accesses per iteration is therefore equal to 24. This number is optimal and can be demonstrated with the following calculations :
 - There is a total of 16 word reads and 8 word writes per iteration for the state arrays. However, as the states are interleaved, the `VLD2.32` instruction leverages 64-bit data transfers between the NEON unit and memory, reducing the number of accesses to 12.
 - There are 4 word reads and 4 word writes involving the filter input with the `VLD1.32` instruction.
 - The bit-mask is written to an array in the stack once per iteration, adding 4 more word writes.

In total, there are therefore 24 accesses per iteration thanks to the powerful `VLD2.32` instruction.

- The number of misses per iteration is fixed at 2.5. This accounts for the two compulsory misses of the states and the compulsory miss of the filter input. As the filter input at a given iteration only uses half of the cache line, the miss happens once every two iterations.
- A periodic pattern is observable in the number of L1 data cache evictions. This is the symptom of cache trashing and is due to the fact that the states `s1` and `s2` are separated in memory. This is further explained in the subsequent paragraph.
- Similarly, this spike pattern is noticeable in the number of stall cycles related with the L1 data cache. In addition, a triangular pattern reappears for sizes multiple of 4096. Stalls have a direct impact on the execution time of the method.

The L1 Data Cache has 32 kB of capacity and 4 ways ; therefore each way therefore has a capacity of 8 kB. The spike pattern in the data cache evictions appears every time the size is a multiple of 1024 ; this corresponds to a 8 kB gap between the the `s1` and `s2` arrays. The `s2` array is accessed two times for read and write operations while the `s1` array is accessed only once for a read. Since the `s1` access appears before the `s2` accesses, the eviction of `s1` does not lead to a conflict miss. However, it induces additional traffic between memory systems in order to take care of the eviction, resulting ultimately in additional stall cycles. Moreover, the triangular pattern is only visible in the measurement of stall cycles, it is therefore due to a variation of the access time of the cache. This is most probably caused by a complex behavior of the cache buffers involving the alignment of memory addresses.

These behaviors cause a considerable problem : the predictable time performances of the algorithm are modified by periodic disturbances and a model architecture optimization based on those results would have to take into account these imperfections. The following solution is therefore proposed to alleviate these imperfections.

5.2.4.3 Memory optimized NEON Intrinsic implementation

First, in order to suppress the periodic data evictions, the memory layout presented in Fig. 5.19 is revised. In addition to the ϵ - ν interleaving of states, blocks of `s1` and `s2` of 8 elements, are interleaved on a second level. This structure is represented in Fig. 5.35.

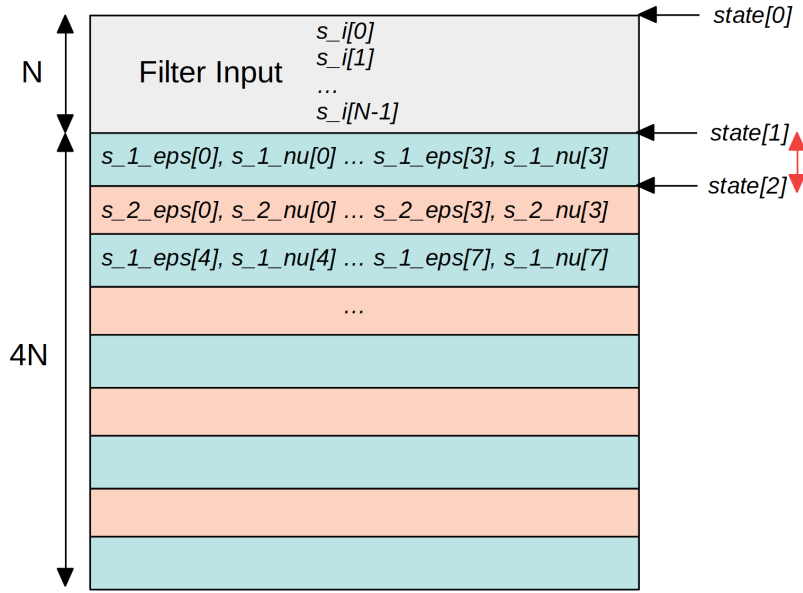


Figure 5.35: Revised layer memory organization to avoid cache trashing.

Secondly, the memory blocks of each layer are aligned in memory with the address of a cache line boundary ; this corresponds to a 32-byte alignment. Therefore, at a given iteration, the `s1` and `s2` states fit inside two consecutive cache lines while the filter input occupies half of a cache line. This modification has two important consequences :

1. The number of cycles that a VLD instruction takes depends on the alignment of the accessed memory block and the presence of the alignment hint `@` in the instruction operand (Section 4.3.3). If the memory is aligned to a 256-bit boundary, one cycle can be spared [95].
2. The cache lines can be pre-fetched using the PLD instruction, allowing to reduce the number of compulsory misses.

The memory blocks are dynamically aligned upon instantiation of the layer, using a customized `struct` in C++. In addition, the GCC routine `__builtin_assume_aligned` is used when declaring the state pointers in the `update` method in order to generate the corresponding alignment hints in the NEON instructions. This last operation is detailed in the snippet of Fig. 5.36.

```

1 // Define alignment hint to a 32-byte boundary
2 #define ASSUME_ALIGNED(type, ptr) reinterpret_cast<type>(__builtin_assume_aligned(↵
   reinterpret_cast<void*>(ptr), 32))
3
4 // Alias states
5 float32_t* si{ASSUME_ALIGNED(float32_t*, state[0])};
6 float32_t* s1{ASSUME_ALIGNED(float32_t*, state[1])};
7 float32_t* s2{ASSUME_ALIGNED(float32_t*, state[2])};

```

Figure 5.36: Memory alignment hint with the GCC builtin routine.

Thirdly, the PLD instructions are leveraged in iteration n to pre-fetch the memory content of iteration $n + m$. Indeed, the memory access time necessary to fetch a cache line might be

larger than the execution time of a loop iteration. In this case it is wiser to pre-fetch the memory content well in advance, at a *distance* $m > 1$. An inline assembly template is used to incorporate Assembly language in C++. This is illustrated in the snippet of Fig. 5.37.

```

1 #define PLD(x)  asm ( "pld [%0]" :: "r"(x))
2
3 // s1 and s2 blocks have a size of 8 (4 eps + 4 nu)
4 // s1 and 2 are interleaved so the next item is at location +16
5 // m argument indicates how far the pre-fetching is performed
6 PLD(s1 + m * 16);
7 PLD(s2 + m * 16);
8 PLD(si + m * 4);

```

Figure 5.37: State and filter input pre-fetching using inline Assembly language.

This third test therefore compares the NEON intrinsic implementation of the second test with the one that revises the memory layout. Several distances of pre-fetching are tested to select the optimal one. The results of this test are shown in Fig. 5.38.

This final test yields even better performances :

- The state interleaving allows to entirely suppress the spikes associated with the data evictions. Furthermore the spikes in the memory stall cycles are also removed thanks to the correct data alignment.
- A large portion of the compulsory cache misses are suppressed with data that is ideally pre-fetched 2 or 3 iterations in advance. However some misses still occur since some data accesses need more cycles to be resolved. This is especially the case when the L2 cache needs to evict cache lines when pre-fetching. A change of slope is readily visible in the number of stall cycles when L2 data eviction occurs for large layer sizes ; it has a small impact in the execution time of the method.
- The miss rate of the method is now around 1% when pre-fetching 2 or 3 iterations in advance.

The memory optimized NEON intrinsic `update` method is therefore our final choice of implementation, with a pre-fetch distance set at $n + 2$ for simplicity. The next test involves measuring the impact the output events on the method's performance.

5.2.4.4 Influence of sparsity

Finally, the influence of the output events is measured by performing computations with non-zero states and inputs. The number of output events is recorded as a fraction of the layer size and referred to as the *spike sparsity* in percents. The impact of the spike sparsity is shown in Fig. 5.39.

The relative impact of additional **STR** instructions due to the output events are relatively low and merely causes an increase of less than 1% for a sparsity below 20%. Their impact can therefore be ignored.

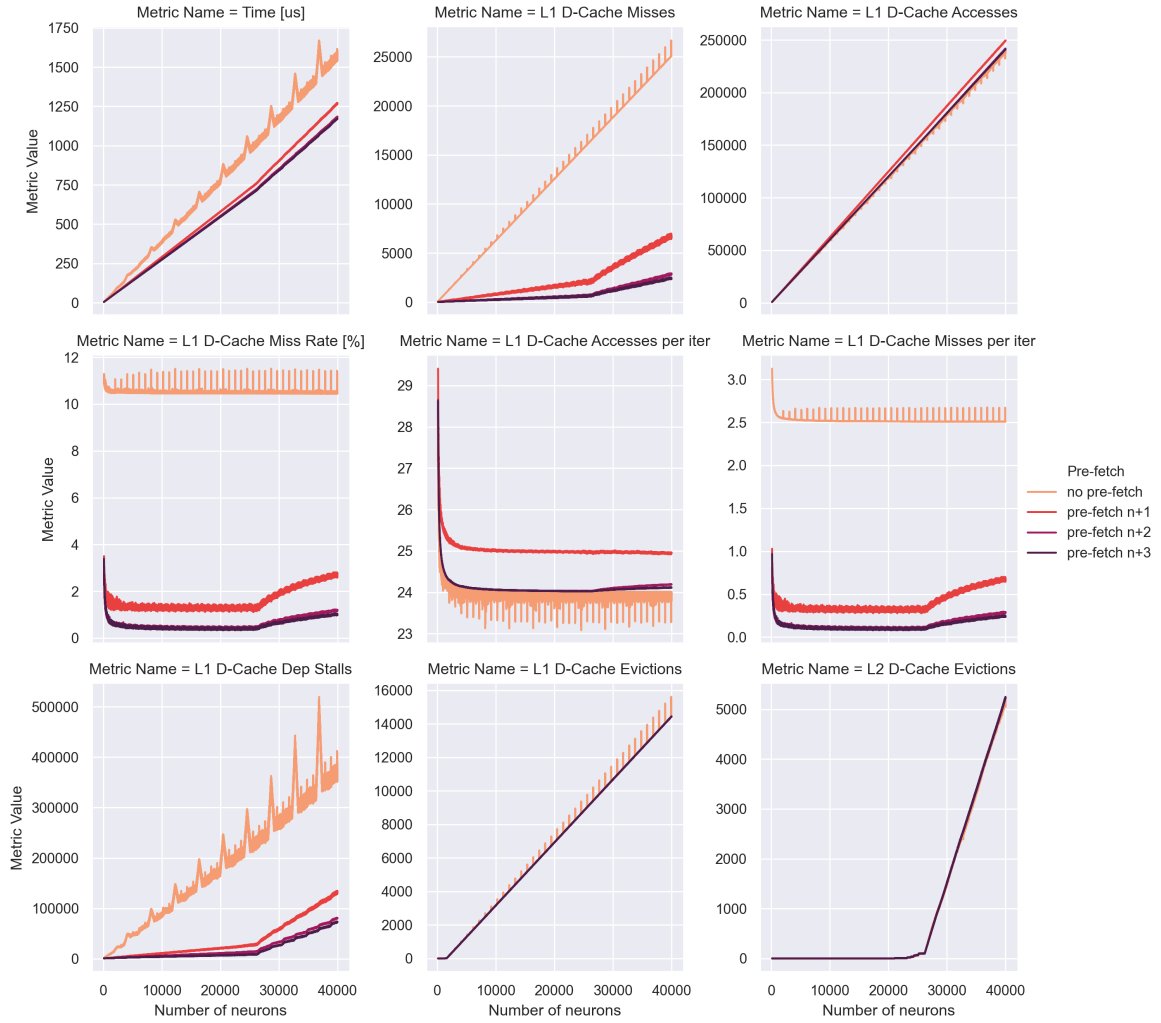


Figure 5.38: Update method metrics comparison between the NEON intrinsic implementation and its memory optimized counterpart, for several pre-fetch distances.

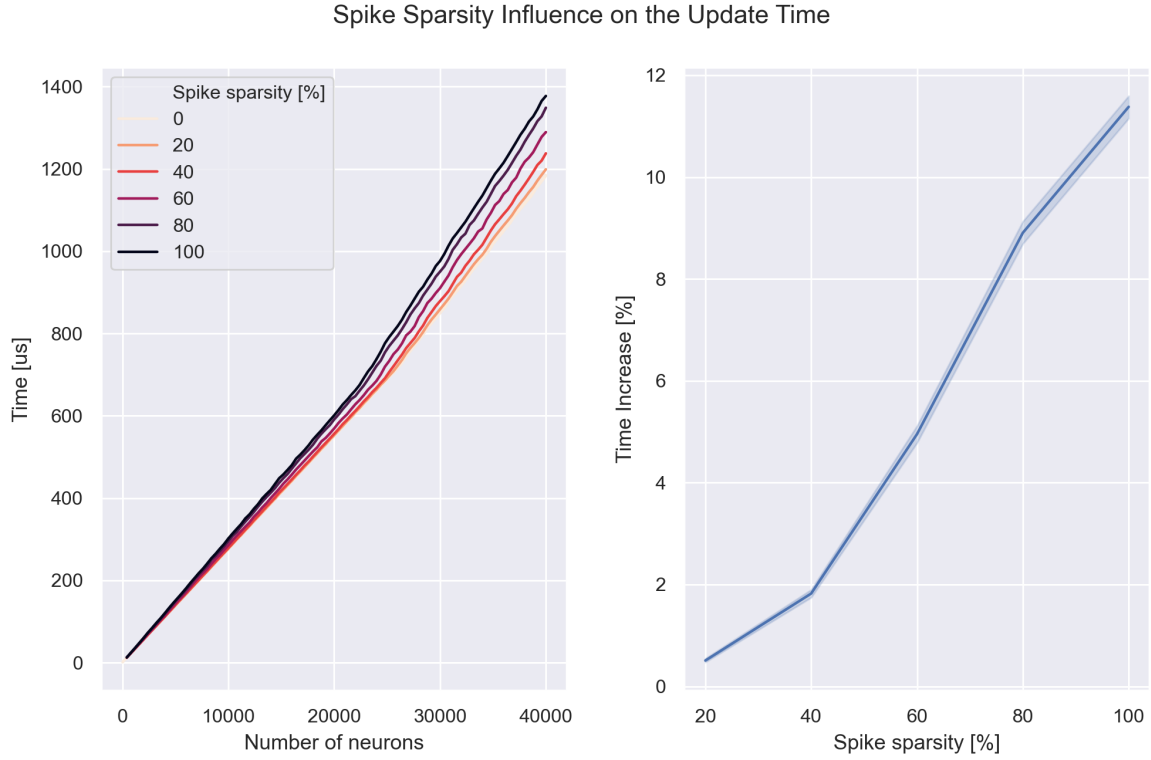


Figure 5.39: Update method output spike sparsity influence. On the left, Thick line : average result over several layer sizes. Shaded area : standard deviation.

5.2.4.5 Comparison of all methods

A synthesized comparison of all implementation variants discussed in this section is displayed below, for a spike sparsity of 0 %. The execution time curves of the methods are displayed in Fig. 5.40. Their slopes are computed in Fig. 5.41 in order to predict the performance of the algorithm for any given layer size with a single coefficient. This coefficient is later referred as the *performance coefficient*.

These graphs underline again the importance of the cache system and the compiler optimization flags. The work of this section has decreased the execution time of the **update** method by a factor of 2.8 and optimized the number of memory accesses to the strict minimum.

5.2.5 Update method optimization for 16-bit floating-point data-types

Another approach to improve the performances of the **update** method is to revise the data-type used for the storage of the filter input and states. The single precision floating-point data-type has a 4-byte width while the half precision floating-point only has a 2-byte width. A cache line can therefore hold twice as many data and the overall program could benefit from a lower miss rate.

The NEON unit offers conversion operations between half precision and single precision formats while the data processing operations are only available for the single precision format.

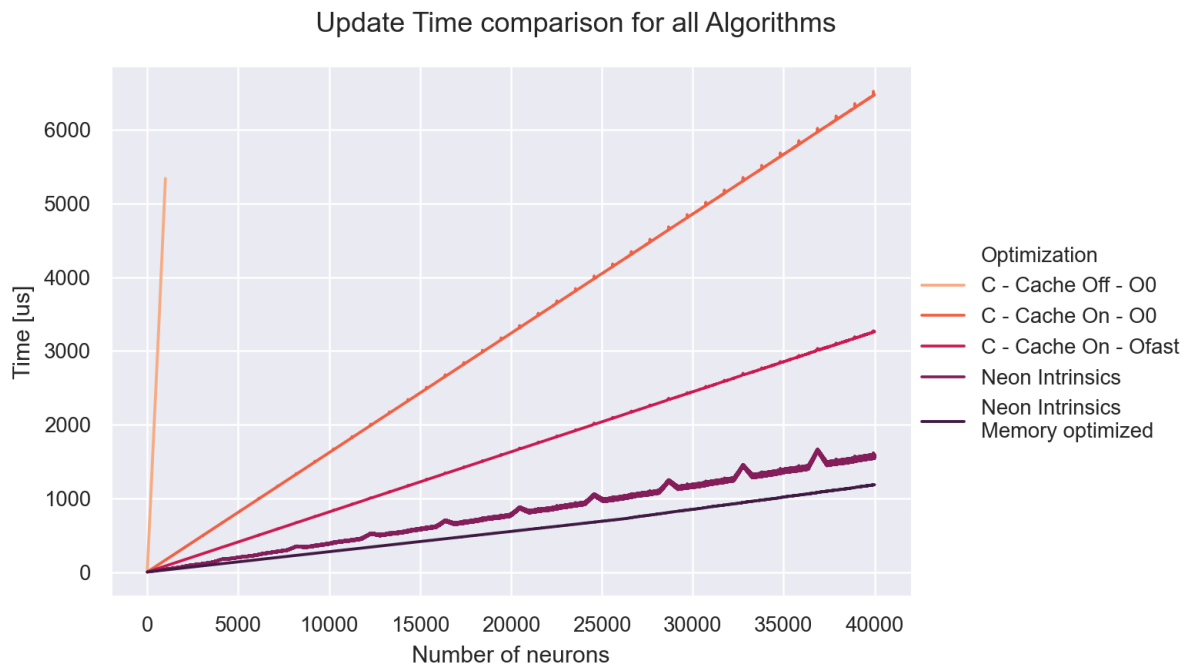


Figure 5.40: update method execution time comparison between all implementations.

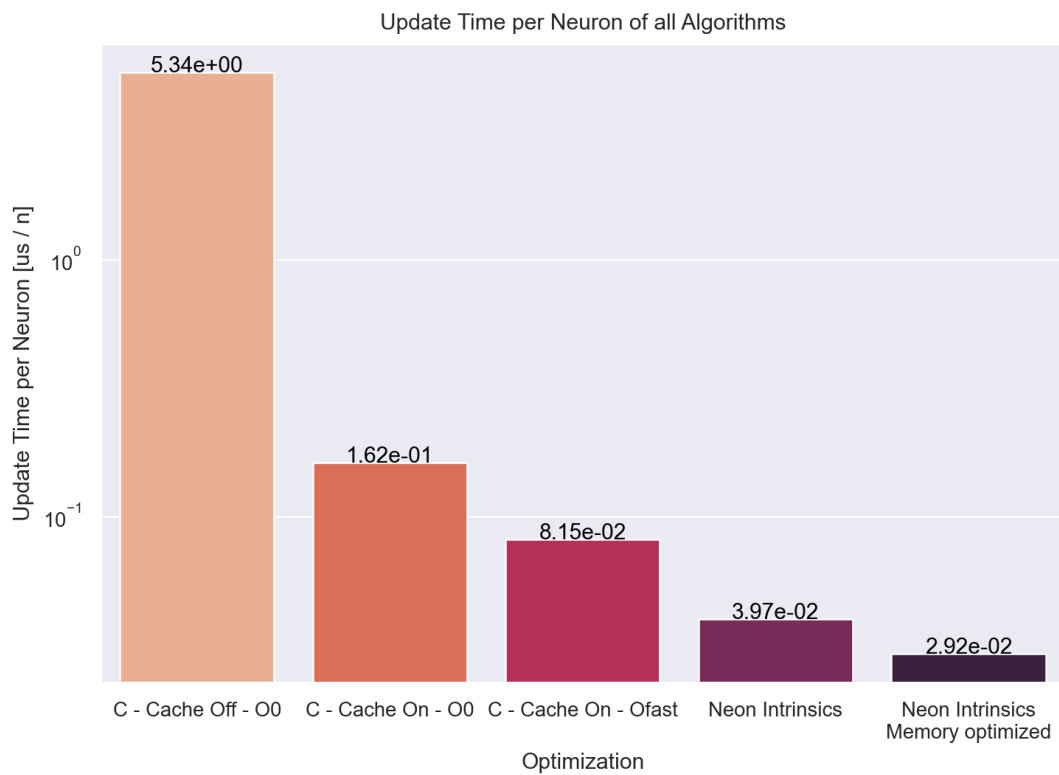


Figure 5.41: update method performance coefficient comparison between all algorithms.

This indicates that we are not able to achieve a higher SIMD parallelization since 32-bit floating-point values are still used at the core of the algorithm. Only the data accesses and cache pre-fetching can have an impact on performance with this approach. Moreover, this impact needs to be large enough to counterbalance the additional cycles required by the conversion instructions.

Building on the memory optimized NEON intrinsic implementation of the last section, additional pieces of code must be integrated to handle the conversion operations. The filter input and states load operations are adapted in Fig. 5.42. The `vld2_f16` is used to perform an interleaved load in 2 D registers. These registers are then expanded to 2 Q registers when using the conversion intrinsic `vcvt_f32_f16`. A similar operation is necessary when loading the loop invariants at the beginning of the function and when storing the new states at the end of an iteration. The complete NEON intrinsic implementation and related Assembly code are shown in Appendix C.1.2.

```

1 // Load states & input
2 float16x4x2_t vs1_f16 = vld2_f16(s1);
3 float16x4x2_t vs2_f16 = vld2_f16(s2);
4 float16x4_t vsinput_f16= vld1_f16(si);
5 vst1_f16(si, zeros_f16);
6
7 // Conversion
8 float32x4x2_t vs1;
9 float32x4x2_t vs2;
10 float32x4_t vsinput;
11 vs1.val[0] = vcvt_f32_f16(vs1_f16.val[0]);
12 vs1.val[1] = vcvt_f32_f16(vs1_f16.val[1]);
13 vs2.val[0] = vcvt_f32_f16(vs2_f16.val[0]);
14 vs2.val[1] = vcvt_f32_f16(vs2_f16.val[1]);
15 vsinput = vcvt_f32_f16(vsinput_f16);

```

Figure 5.42: Conversion operations from 16-bit floating-point to 32-bit floating-point data-types for the filter input and state load.

This implementation is tested in the same test-bench as the last section for comparison with the 32-bit implementation ; the results are shown in Fig. 5.43. These measurements call for the following comments :

- The number of accesses per iteration is reduced from 24 to 16.
- The number of misses and memory stalls is reduced by a factor of 2.
- Since the data-type takes two times less space in memory, data evictions for the L1 cache start at 3246 neurons and the data evictions for the L2 cache start at 52429 neurons. This result is interesting as the layer data is now more susceptible to stay in the cache in-between function calls.
- The execution time of the 16-bit method however is slightly longer than the 32-bit one. This is due to the additional conversion instructions `VCVT` that take 2 cycles to execute in the NEON pipeline and up to 6 cycles if there is a dependency stall.

The half precision floating-point `update` method is therefore not competitive in terms of execution time but rather as regards memory usage and the number of memory accesses.

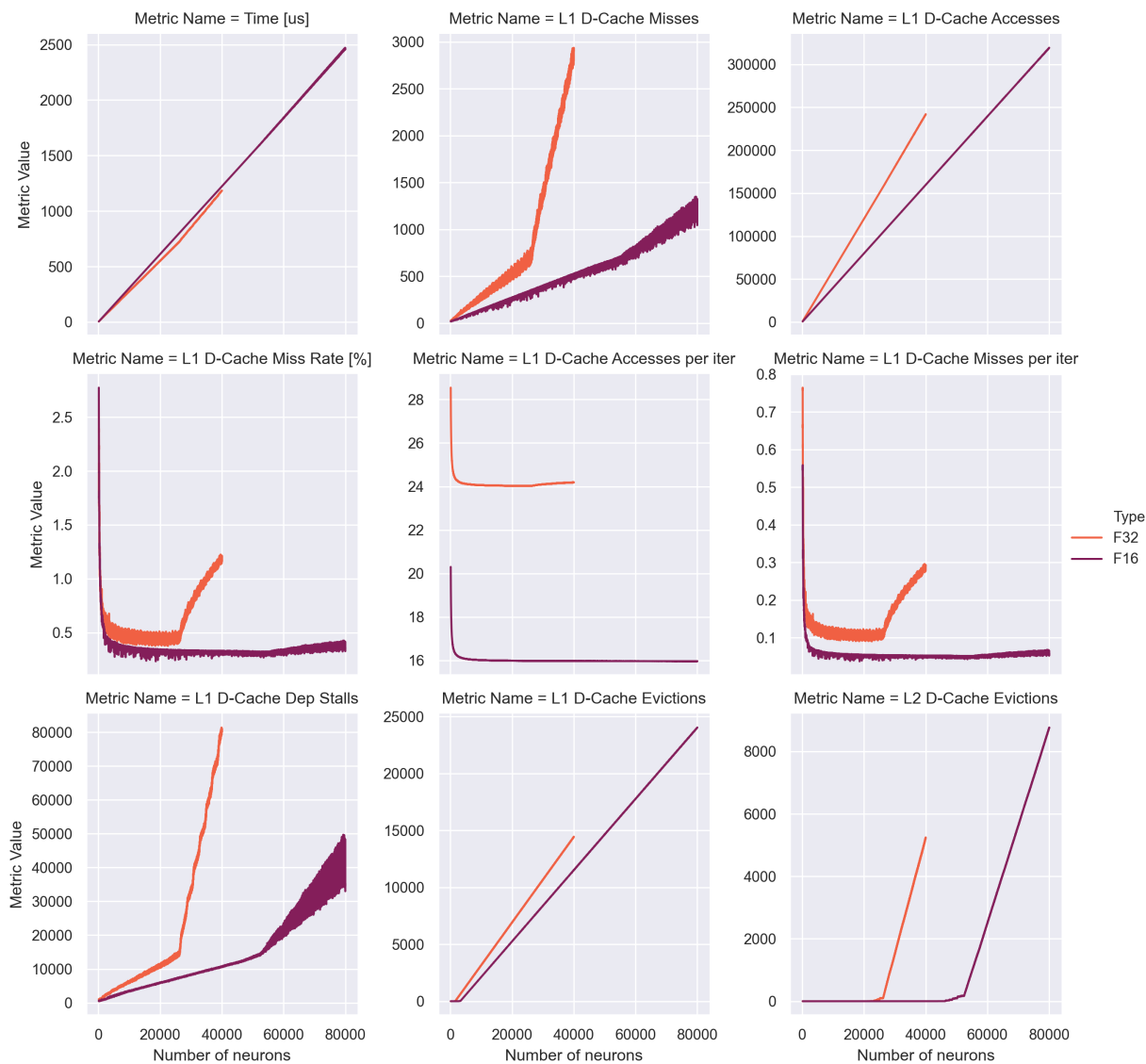


Figure 5.43: `update` method metrics comparison between the memory optimized 16-bit and 32-bit floating-point NEON intrinsic implementations.

It could therefore offer a low-power solution as accesses to main memory are reduced and the memory capacity needs are divided by 2. Moreover, it could potentially better cache the layer data in-between time-steps. The performance coefficients of the 32-bit and 16-bit methods are shown in Fig. 5.44.

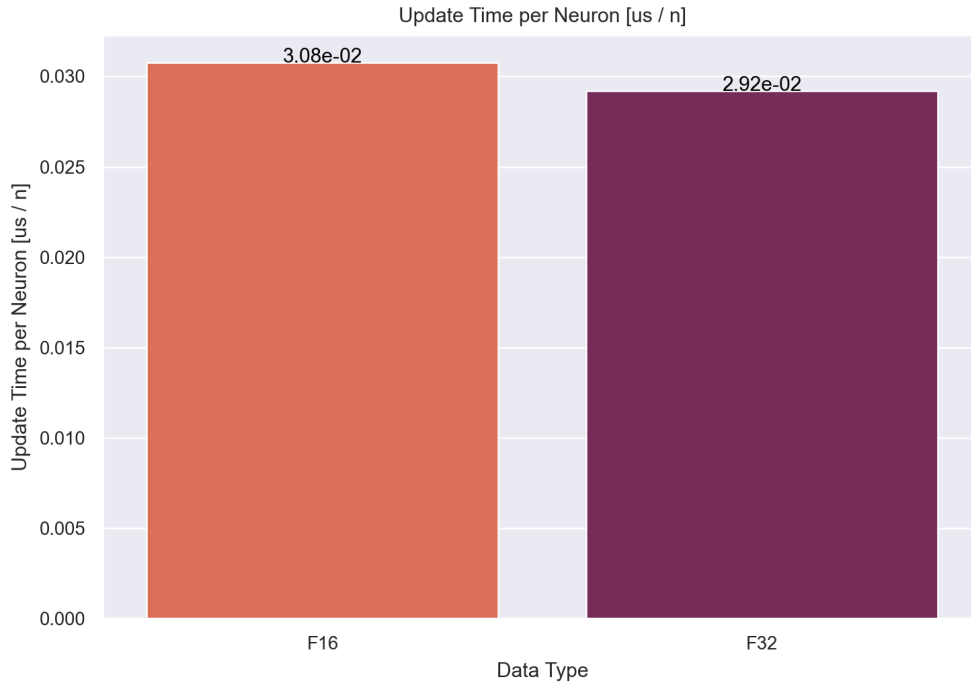


Figure 5.44: `update` method performance coefficient comparison between the memory optimized 16-bit and 32-bit floating-point NEON intrinsic implementations.

5.2.6 Layer operations optimization for 32-bit floating-point

The second way we can improve the overall performance of the `forward` method is to enhance the `layer_operation` methods. However, this enhancement can only offer a limited improvement since the layer methods only accounts for around 15.1 % of the total forward execution time. This section therefore brings a faster analysis, the methods are upgraded to NEON intrinsics and immediately improved using the concepts discovered in the last section such as memory alignment and pre-fetching. The auto-vectorizer approach is not tested again as the intrinsics are easy to use and have proven to offer a better control on the produced instructions. All the method's implementations can be found in Appendix C.2 and C.3.

5.2.6.1 Dense optimization

The inner loop of the snippet already presented back in Fig. 5.9 is improved by replacing its statements with NEON intrinsics. In order to leverage the 64-bit data transfers of the `VLD2.32` instruction, both the filter input and layer weights are fetched by de-interleaving with the `vld2q_f32` intrinsic. Moreover, it allows to use one cache line per iteration as 8

floating-point elements are processed at a time. Since the relative position of the inputs and the coefficients are not modified, the interleaving does not have any impact on the result.

The registers are then added together and stored back into the filter input array with interleaving using the `vst2q_f32` intrinsic. Additionally, the filter input and weight arrays are aligned to a cache line and PLD instructions are added at the beginning of each loop iteration to pre-fetch the input and weight cache lines with a distance of 1 iteration.

At last, the `__restrict` keyword is used in front of the pointers in the argument list to provide additional informations to the compiler [96]. It indicates that the memory regions addressed by the pointers do not overlap with each other ; a phenomenon also known as *pointer aliasing*. This keyword can lead to significant performance improvements when one array is filled with values computed from a second array. Once the `__restrict` keyword specified, the compiler does not need to store a copy of the source array when computing the values to be written to the destination array. These modifications are displayed in Fig. C.8.

One last remark should be made before the result analysis : the number of channels of any dense layer is now restricted to a multiple of 8. This optimization is therefore not applicable to the last dense layer as its number of layers is equal to the number of classes predicted by the model ; a separate method must be used for this case. This is easily achievable in C++ thanks to the template meta-programming paradigm.

A new test-bench is created to measure the performances of the C and NEON intrinsic dense implementations. It measures the execution time of the `dense_operation` method for different input and output number of channels. When testing a layer with n input channels, the number of spikes at the input is swept from 0 to $n/2$ since the Slayer model obtained in 3.4.4 shows sparsity values in this range. As in the last test-bench, the results are averaged over 10 trials and the cache system is purged at each trial. The results of the test-bench are shown in Fig. 5.45 for different numbers of input and output channels.

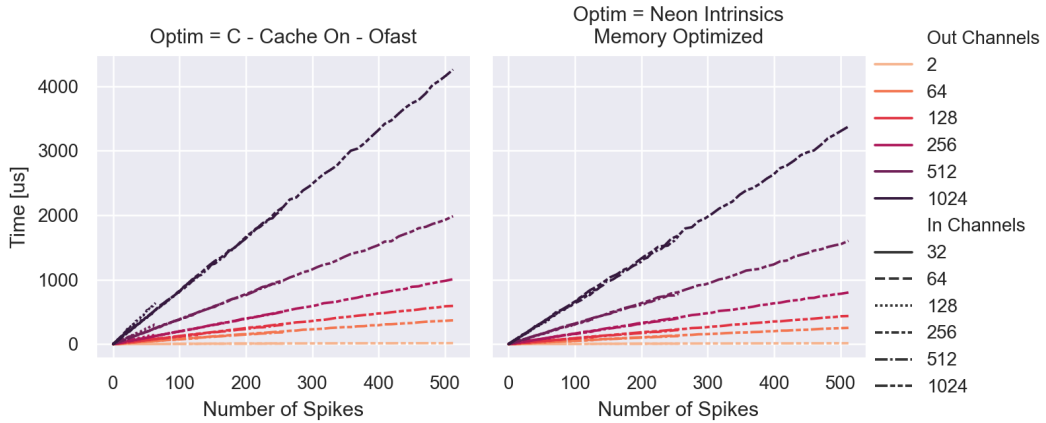


Figure 5.45: Input layer size influence on the dense method execution time C and NEON implementations.

As already predicted in the FLOP analysis in Eq. 5.18, the input channel size has little influence on the execution time which is linear with respect to the number of events. The NEON intrinsics offer a limited boost in performance compared to the C implementation.

The influence of the input channels is averaged in Fig. 5.46 and the slope of each curve is

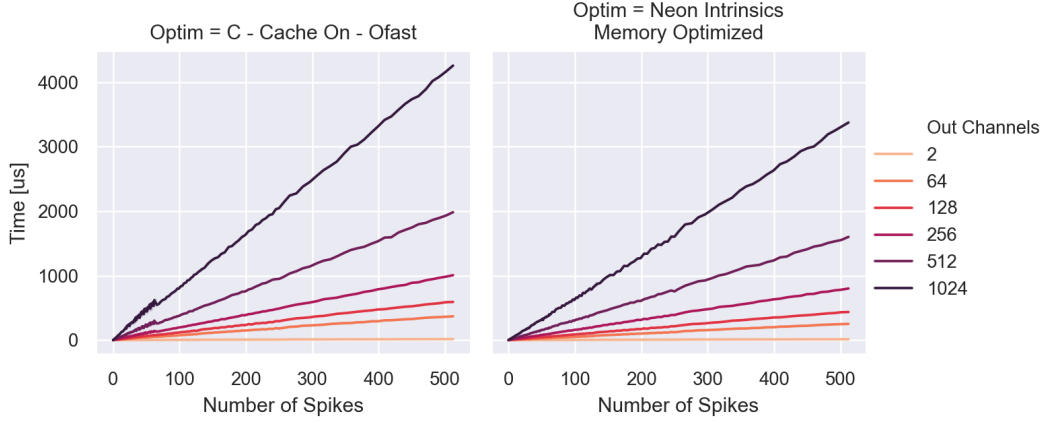


Figure 5.46: Dense method execution time vs. number of spikes for C and NEON implementations and multiple output channels.

drawn in terms of the output number of channels on the left of Fig. 5.47. Once more, this graph shows that the execution time is linear with respect to the number of output channels, as predicted in Eq. 5.18. Finally, the slope is extracted again on the right side of Fig. 5.47 to synthesize the performance into a single coefficient. This quantifies an 18% reduction in execution time between the C and NEON intrinsic implementations.

5.2.6.2 Convolution optimization

The adaptation of the `conv_operation` method essentially follows the same steps as the one for the dense layer. The innermost loop of the algorithm of Fig. 5.14, corresponding to the output channel dimension, is re-written with NEON intrinsics to use the interleaved loads and store instructions. In addition, the input and weights are pre-fetched for the next iteration. As in the dense layer, the number of output channels is now restricted to a multiple of 8. This is shown in the small `vadd` inline function in Fig. C.14.

A last test-bench is created to measure the performance of the method for different number of input and output channels. The test uses a kernel size of 5×5 and two input spatial sizes are tested : 53×41 and 25×19 . They are later referred to as the *input format*. These choices correspond to the parameters of the optimized Slayer model. The number of spikes generated is chosen to lie between 0 and 10% of the input spike since this level of sparsity in the convolutional layers has been observed for the selected model.

As shown in Fig. 5.48, the execution time is not influenced by the number of input channels and their impact can be averaged. However this graph only shows the results for the 53×41 input format. The influence of the input format is depicted in Fig. 5.49

This influence can be explained by the border effects explained in Section 5.1.3. Since the 25×19 format has a greater chance to have input events that are close to a border, the truncation of the filter happens more frequently and results in less operations per event.

The superiority of the NEON intrinsic implementation is readily visible and is quantified by taking the slope of the curves for each output channel and input format. This computation is displayed in Fig. 5.50, the execution time per spike is linear in terms of the number of output

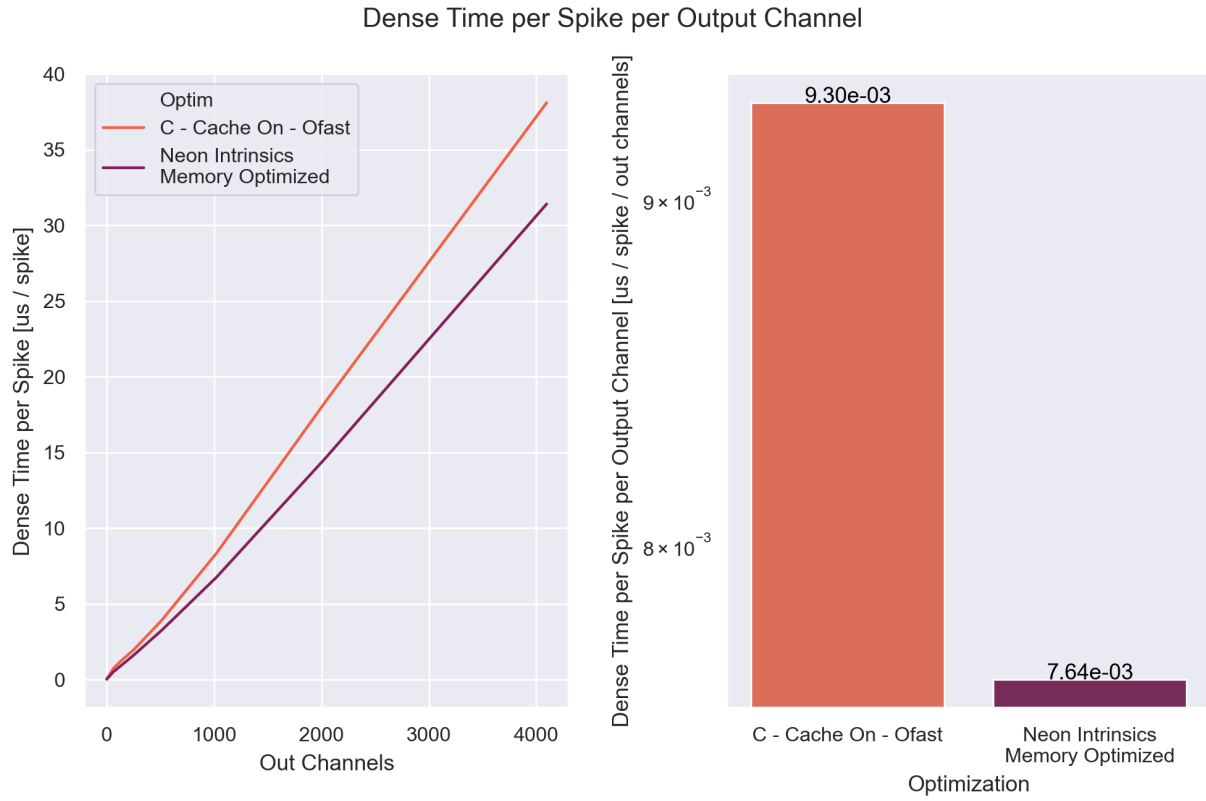


Figure 5.47: Dense method execution time per spike and per output channel for C and NEON implementations.

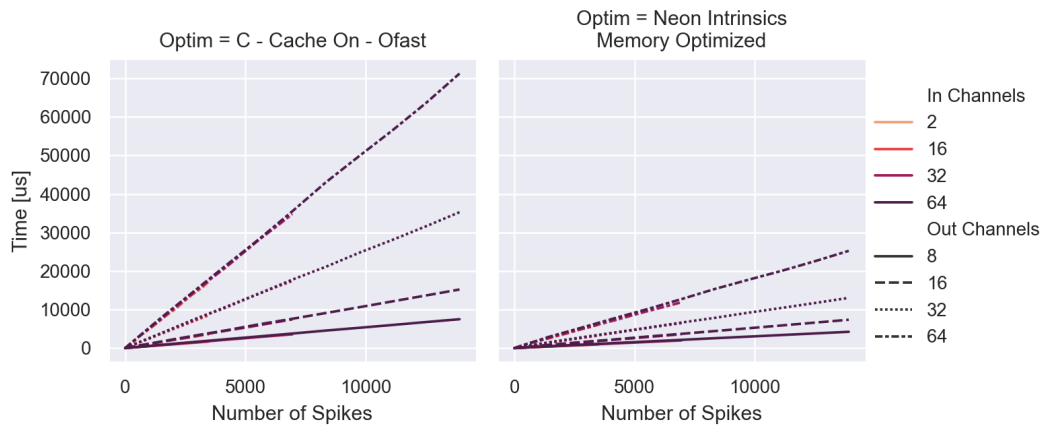


Figure 5.48: Convolution method execution time vs. number of spikes for C and NEON implementations and multiple input and output channels.

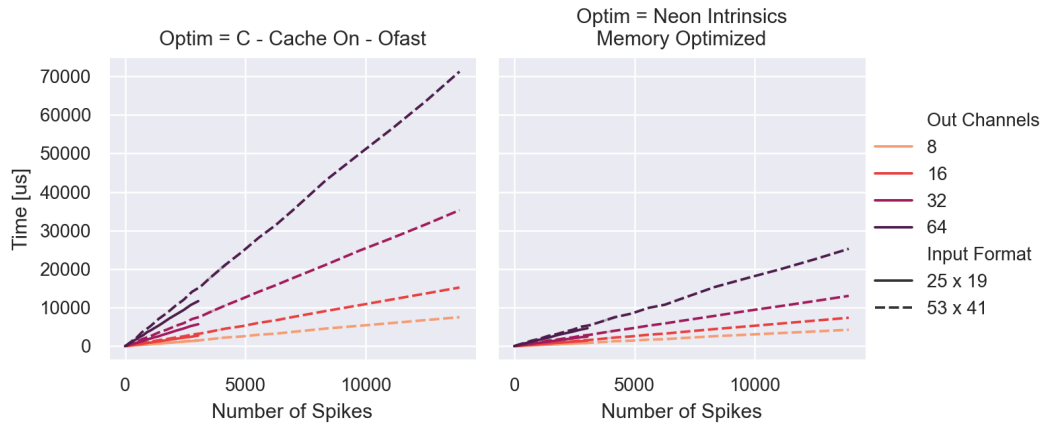


Figure 5.49: Convolution method execution time vs. number of spikes for C and NEON implementations and multiple output channels and input formats.

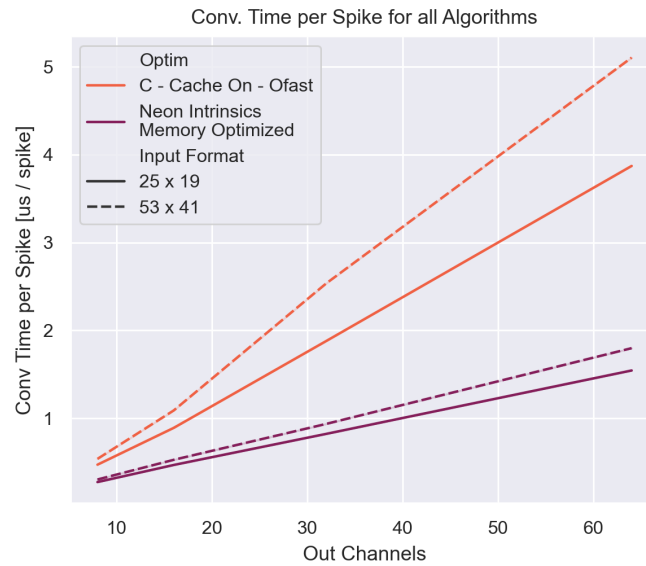


Figure 5.50: Convolution method execution time per spike vs. number of output channels for C and NEON implementations and multiple input format.

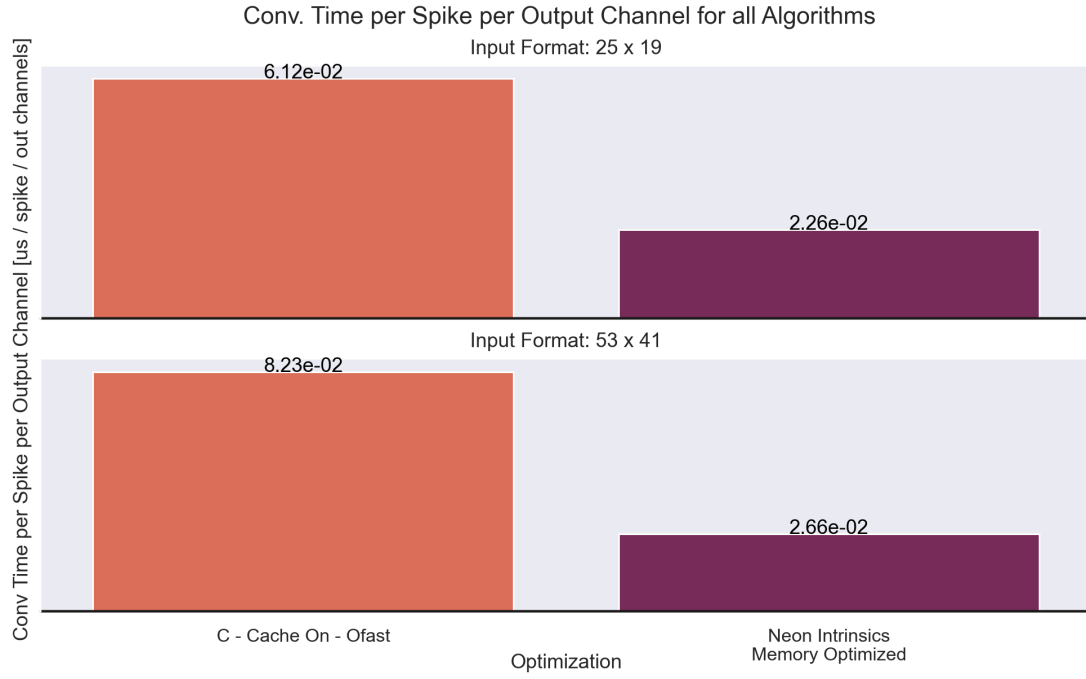


Figure 5.51: Convolution method execution time per spike and per output channel for C and NEON implementations and input formats.

channels, as predicted in the FLOP analysis in Eq. 5.41. Finally, the slope is computed again in Fig. 5.51 to synthesize the performance into a single coefficient. This quantifies on an average a 65% reduction in the execution time between the C and NEON intrinsic implementations.

5.2.7 Layer operations optimization for 16-bit floating-point

The `layer_operation` methods can also be modified to work on half precision floating-point data-types. However, it requires the offline conversion of the weight tensors coming from the Pytorch framework. The lookup table based method described in [105] is used for that purpose, it involves the following conversion rules :

- Really small F32 values and F32 zero map to the F16 zero.
- Small F32 values map to subnormal F16 values.
- Regular magnitude F32 values simply lose precision when converted into F16 values.
- Large F32 values map to the F16 infinity.
- Infinity and NaN F32 values remain the same.

This method respects the IEEE-754 standard and the conversion instructions of the NEON unit will recover the corresponding values in 32-bit with a loss of precision.

5.2.7.1 Dense optimization

In order to avoid unnecessary VCVT operations, the loop order of the code in Fig. C.8 is inverted. The outer loop works on tiles of 16 elements of the filter input array while the inner loop skims across the event indices. The filter input elements are therefore converted only once in the outer loop and the weight elements are converted for each input event. This process is illustrated in the snippet of Fig. C.10.

As lines of 16 elements are fetched with the `vld2q_f16` intrinsic, the processing is duplicated into four 4-lane operations. In the worst case scenario where the compiler do not re-use any register, the algorithm takes at most 15 Q registers. This means that in the worst case, the number of registers necessary are still below the limit and the algorithm does not need to push registers on the stack. A comparison of the performance coefficient is shown in Fig. 5.52.

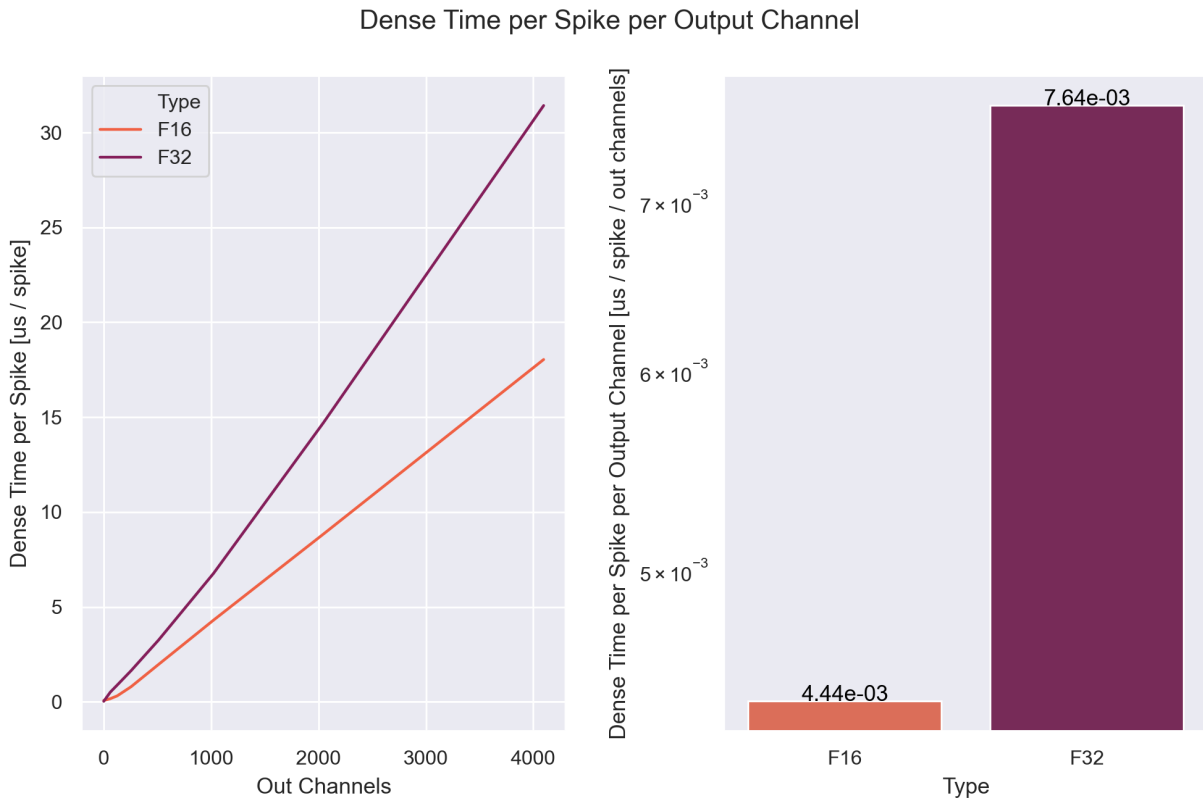


Figure 5.52: Dense method execution time per spike and per output channel for half and single precision data-types.

This experiment shows that the 16-bit floating-point implementation outperforms the 32-bit one, with a 58% reduction.

5.2.7.2 Convolution optimization

Unfortunately, the `conv_operation` method cannot benefit from the same reordering as in the `dense_operation` method. As shown in Fig. C.16, each filter input and weight elements in the inlined `vadd` function must be converted at each iteration of the output channel loop.

Load and store instructions with interleaving are used to fetch data in 2 Q registers. The filter input and weights are pre-fetched in the output channel dimension. The performance coefficients are displayed in Fig. 5.53.

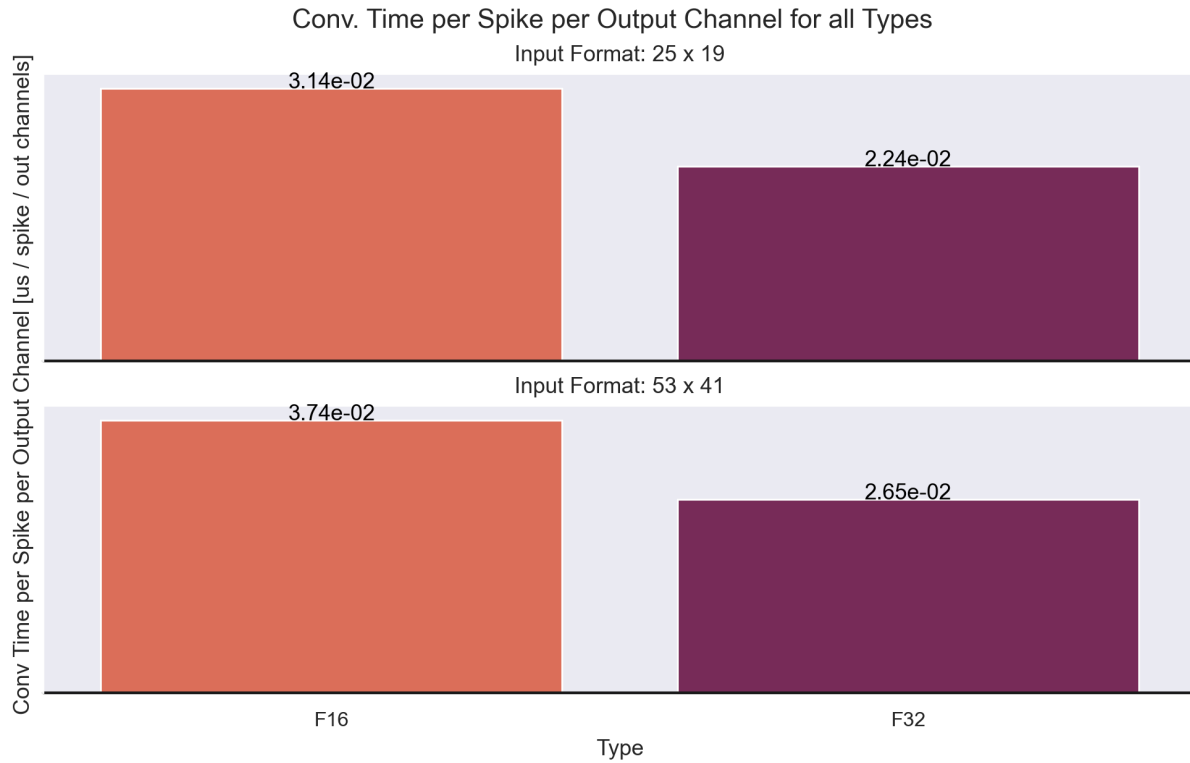


Figure 5.53: Convolution method execution time per spike and per output channel for half and single precision data-types.

This time the `conv_operation` method for 16-bit floating-points takes longer than its 32-bit counterpart. The additional cycles necessary to perform the conversion instructions are not counterbalanced by the reduction in cache accesses. The increase in execution time is approximately 40%.

5.2.8 Performance summary

To conclude the performance analysis, a summary table is provided in Table 5.2. Results are re-written in nanoseconds to obtain a better rounding of the coefficients.

Method	F32 Coefficient	F16 Coefficient	Unit
Update	29.2	30.8	[ns / neuron]
Convolution	26.6	34.4	[ns / (spike · output channel)]
Dense	7.6	4.4	[ns / (spike · output channel)]

Table 5.2: Performance coefficients for the optimized methods.

Combining these results with the statistics of the model in Section 3.4.4, the optimized model `forward` method is predicted to have the average execution time specified in Table 5.3. If confirmed, this result would mean that the method meets the real-time requirements.

F32 Time/time-step [us]	F16 Time/time-step [us]
1400	1460

Table 5.3: Estimated execution time of the optimized model.

This time will most probably be larger in practice since the behavior of the `Model` class that combines the different layers has not been taken into account. Penalties associated with function calls and branch miss-predictions will have an influence. However, the 16-bit floating-point implementation could show better performances by leveraging data re-use in-between time-steps as the caches are more susceptible to keep the data.

For this reason, the next section profiles the complete program with the same methodology as in Section 5.2.2. Additionally, the model’s output is compared with the Pytorch Framework to track variations in the model’s accuracy.

5.3 Model validation of performances and accuracy

The goals of this second profiling are twofold. It is aimed at identifying the new bottleneck in timing performances and assessing the quality of the layer metrics proposed in Table 5.2. The profiling is realized using the same 36 samples as in the initial profiling, with the Barman library.

5.3.1 Barman profiling for the 32-bit floating-point implementation

The timing results are displayed in Fig. 5.54 to 5.57 for the F32 data-type. This analysis reveals several changes in comparison with the initial profiling :

- The execution time of the `update` method has a standard deviation of at most 8 μs . This is a significant improvement due to the optimized memory organization (Fig. 5.54).
- The execution time of the `update` method is no longer dominant anymore in every layer. The second and third layers have `conv_operation` and `dense_operation` methods that can take more time to execute than the `update` method (Fig. 5.54 and 5.57. The layer update now counts for 70.5% of the total execution time (Fig. 5.56).
- The average execution time of the forward method is 1540 μs , with a 75% quantile at 1700 μs and a maximum recorded at 2350 μs (Fig. 5.55).

The average execution time of the forward method is therefore reduced by 63.3% compared to the initial solution. The performance coefficients in Table 5.2 are derived from the observations in order to validate their accuracy when operating the program on actual samples, accounting for the long term effects of the cache system. The observed coefficient

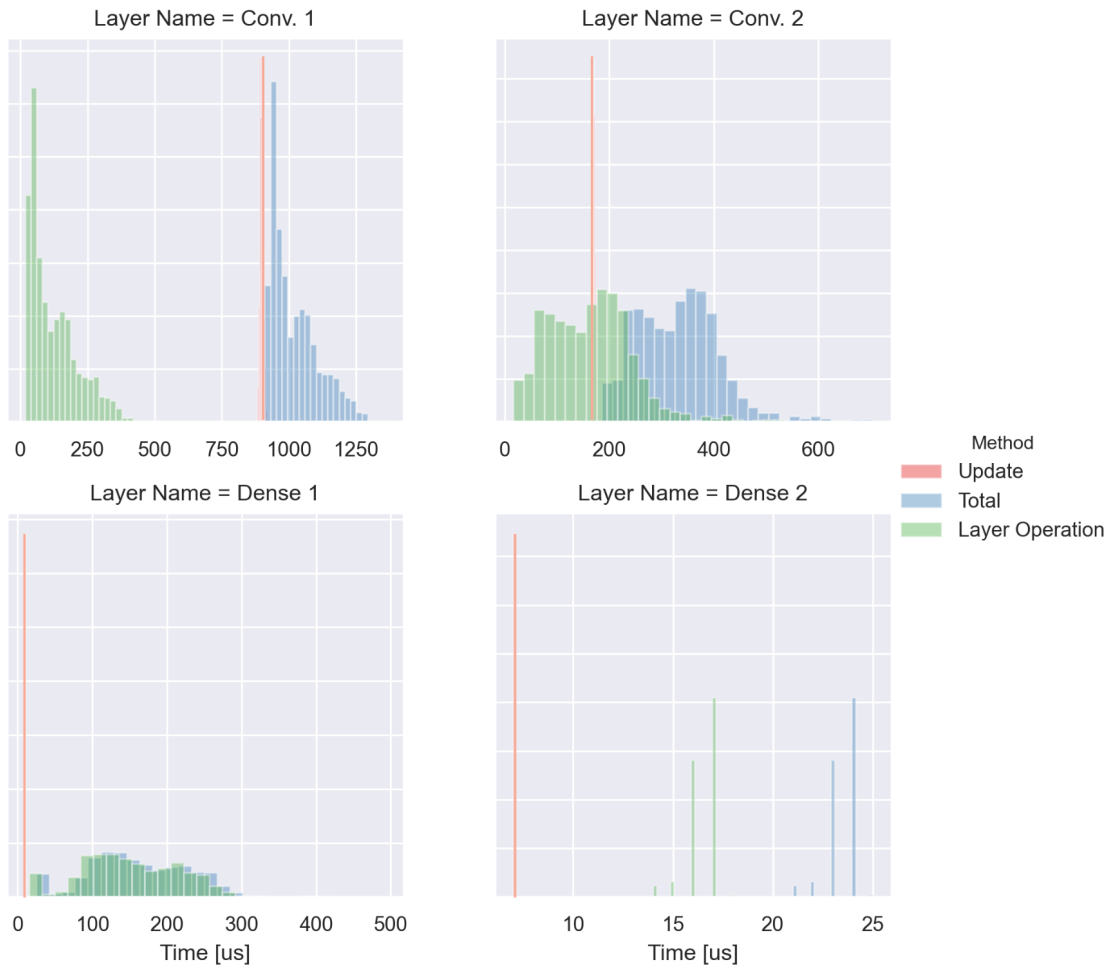


Figure 5.54: Execution time distribution per layer for the F32 NEON intrinsic implementation.

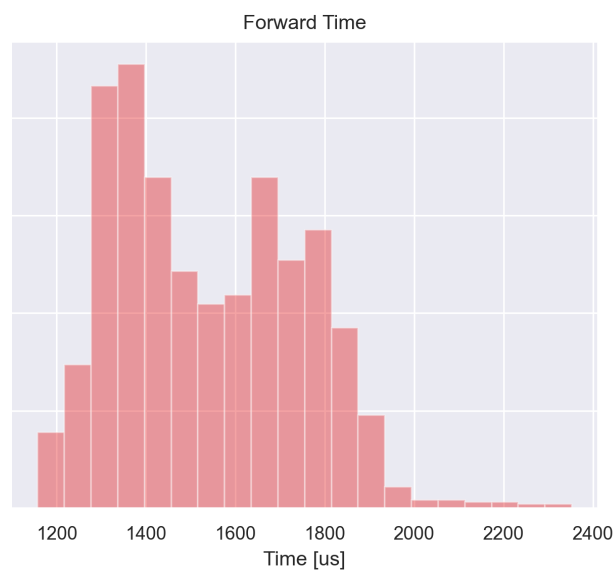


Figure 5.55: Execution time distribution of the **forward** method for the F32 NEON intrinsic implementation.

Average time repartition for methods

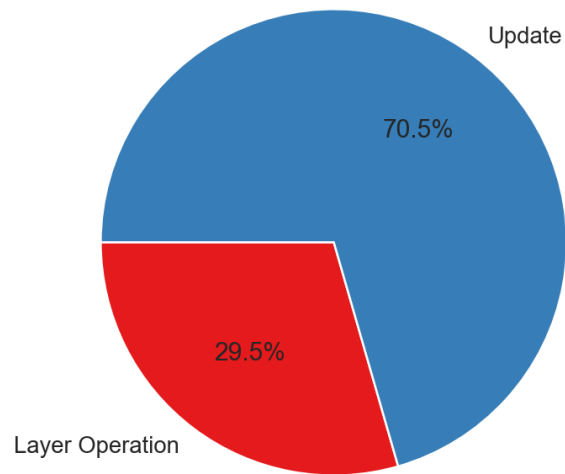


Figure 5.56: Average execution time repartition for the `layer_operation` and `update` methods of the F32 NEON intrinsic implementation.

Average time repartition for layers and methods

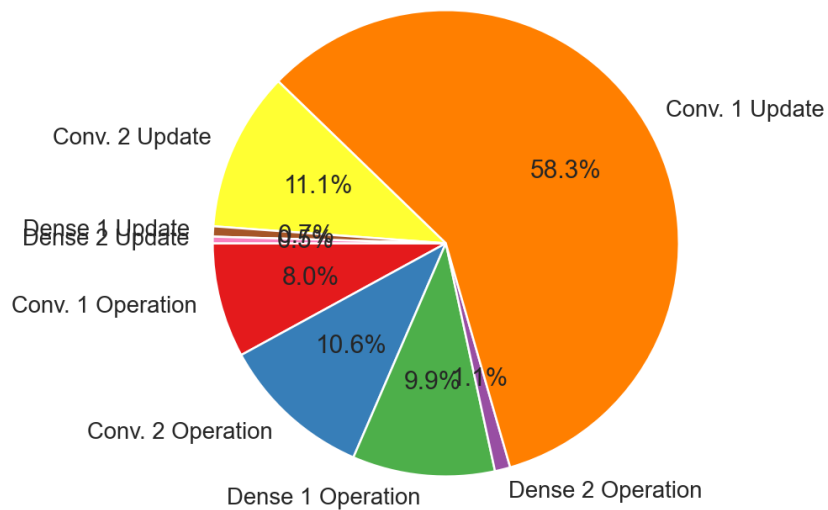


Figure 5.57: Average execution time time repartition for the different layers and methods for the F32 NEON intrinsic implementation.

for the **update** method on the second profiling is identical to the one of the test-bench, it is displayed on the right of Fig. 5.59.

Concerning the layer operations, the execution time is more fluctuating and the recorded samples are shown in Fig. 5.58. This behavior is most probably due to the cache systems and the fact that the weights are randomly accessed depending on the event indices. Data could therefore rest in the cache between time-steps. The slopes corresponding to a least-square fit are computed in Fig. 5.59.

The value of the observed coefficients for the **conv_operation** increases by a 30 % factor compared to the values of Table 5.2. Similarly, the **dense_operation** execution time deviates by 8 %. Nevertheless, as the **update** method is accountable for 70 % of the total execution time, the prediction made in Section 5.2.8 regarding a 1400 μ s mean execution time remains quite accurate.

At last, the PMU recorded metrics for the entire program are computed in Fig. 5.60. The data accesses and data miss graphs are correlated with the number of input spikes in each layer for each time-step. The execution time therefore also follows this trend and the maximum execution time appears after a few milliseconds when the number of spikes is at its peak.

The program shows a remarkably low miss rate of 2 % which appears to be the lowest for convolutional layers having the fewest input spikes. The number of instruction misses was not taken into account during the various optimizations in the last sections. We indicate here that it is comparatively negligible compared to the data misses.

5.3.2 Barman profiling for the 16-bit floating-point implementation

Finally, the results for the F16 data-type are shown in Fig. 5.61 to 5.64.

This analysis reveals several changes compared to the F32 solution :

- The execution time of the **update** method has a standard deviation of at most 1 μ s (Fig. 5.61).
- The **update** method accounts for 76.2% of the total execution time (Fig. 5.63).
- The average execution time of the **forward** method is 1480 μ s, with a 75% quantile at 1620 μ s and a maximum recorded at 2220 μ s (Fig. 5.62).

The performance coefficients are derived from the measurements in Fig. 5.66 and only deviate by a few percents compared to the test-bench values. This is visible in Fig 5.65 where the measurements are a lot more concentrated around the fitting line.

At last, the PMU metrics indicate in Fig. 5.67 an overall number of data accesses divided by two, compared to the F32 implementation. The data cache miss rate is below 1% on average and the number of cache dependency stalls is divided by 5.

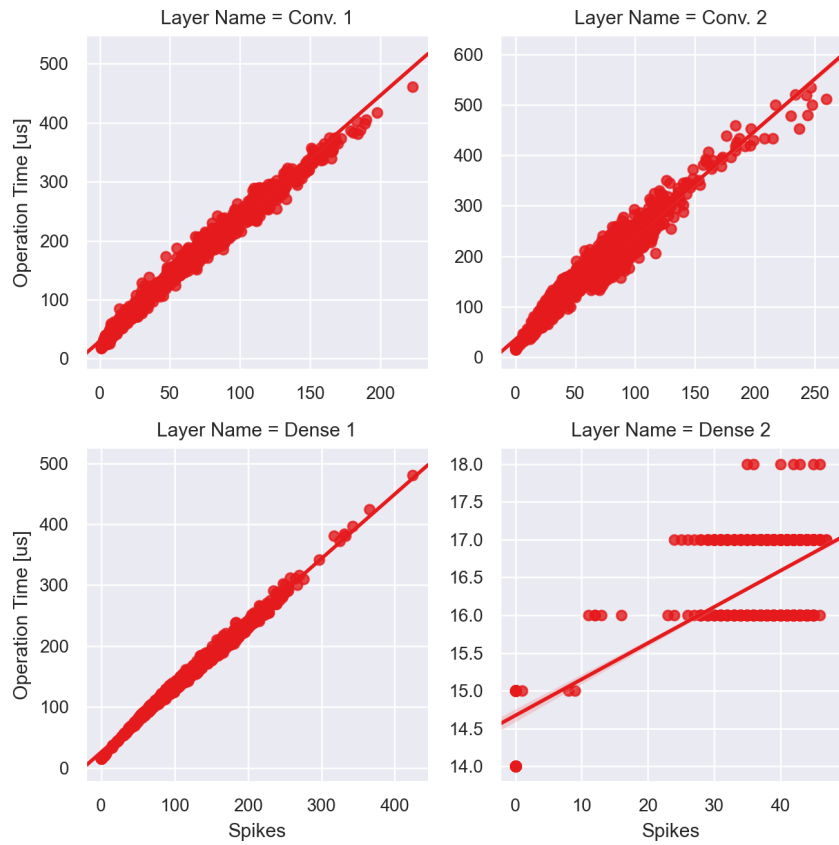


Figure 5.58: Observed characteristics of the layer operation execution time vs. the number of spikes in each layer for the F32 data-type.

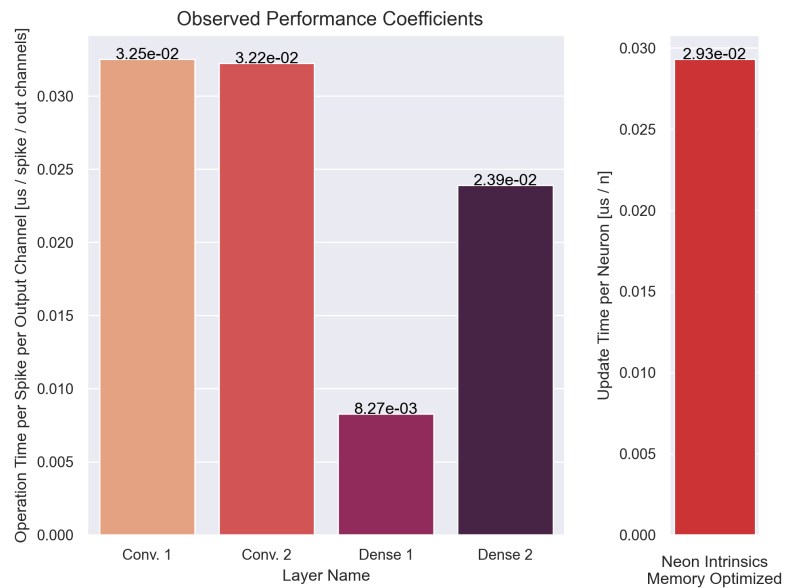


Figure 5.59: Observed performance coefficients during the model's operation for the F32 data-type.

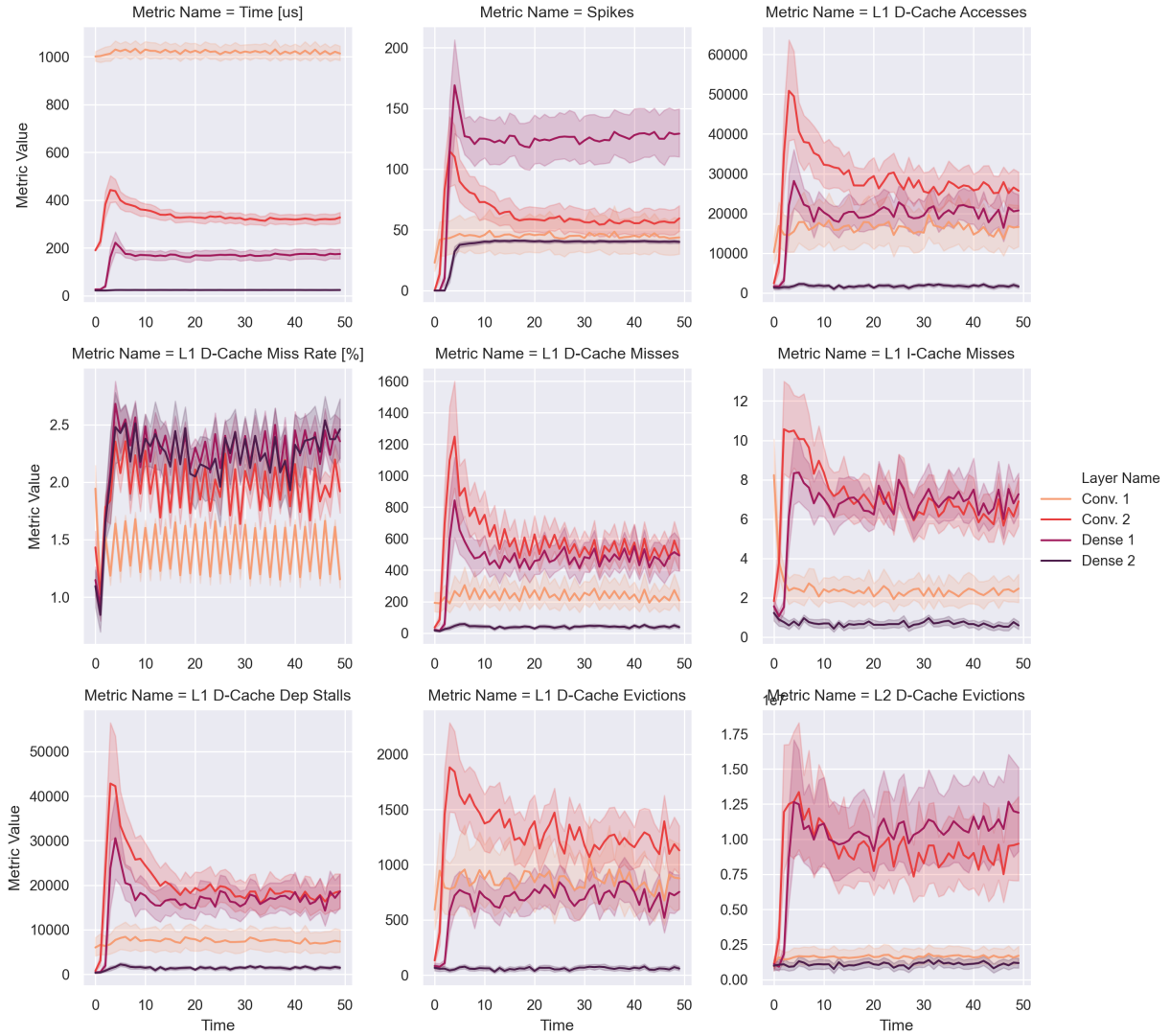


Figure 5.60: **forward** method metrics for the F32 implementation for each time-step, averaged over 36 samples. Thick lines indicate the average and shaded areas represent the standard deviation.

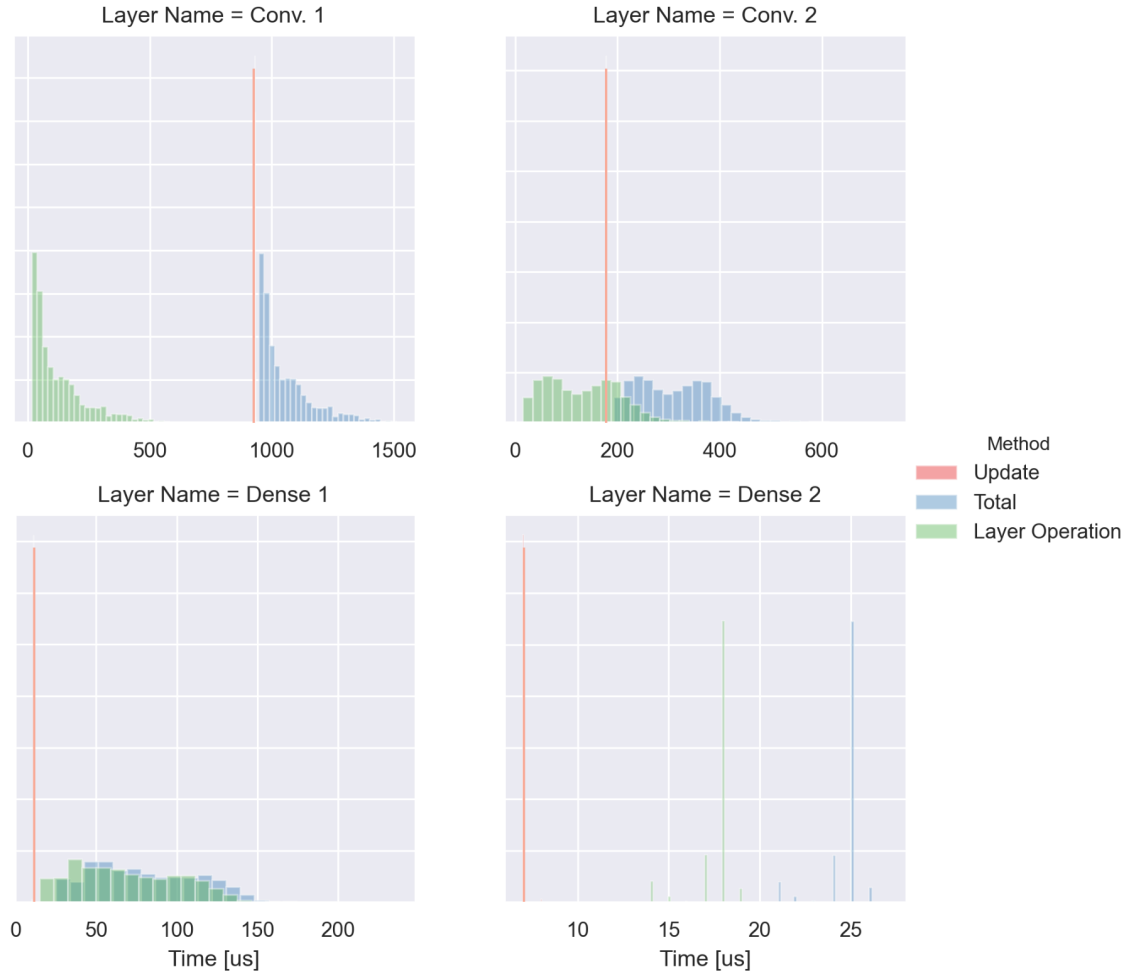


Figure 5.61: Execution time distribution per layer for the F16 NEON intrinsic implementation.

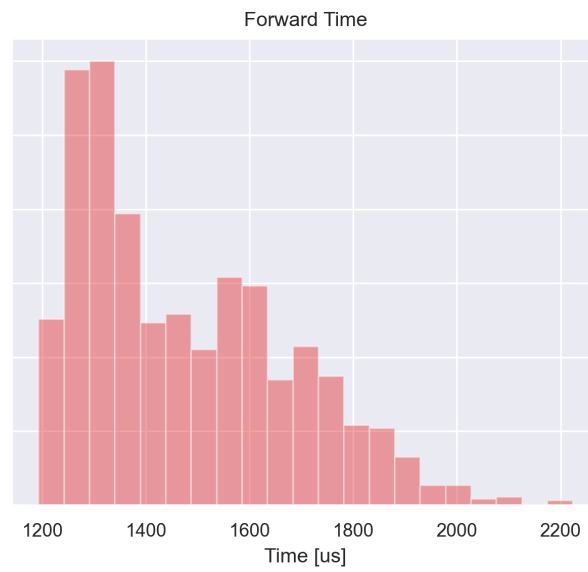


Figure 5.62: Execution time distribution of the **forward** method for the F16 NEON intrinsic implementation.

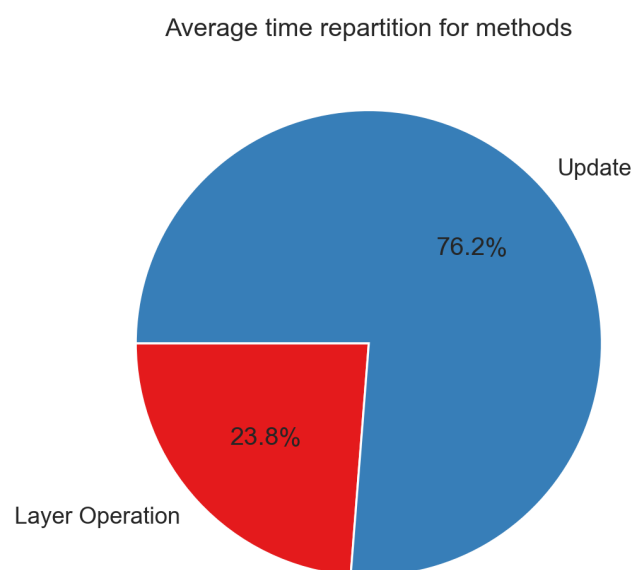


Figure 5.63: Average execution time repartition for the `layer_operation` and `update` methods for the F16 NEON intrinsic implementation.

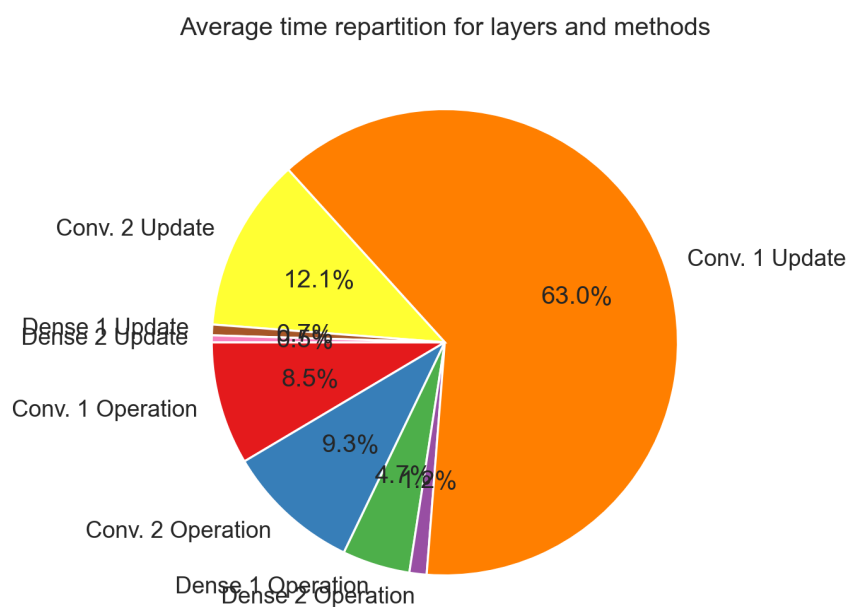


Figure 5.64: Average execution time time repartition for the F16 NEON intrinsic implementation.

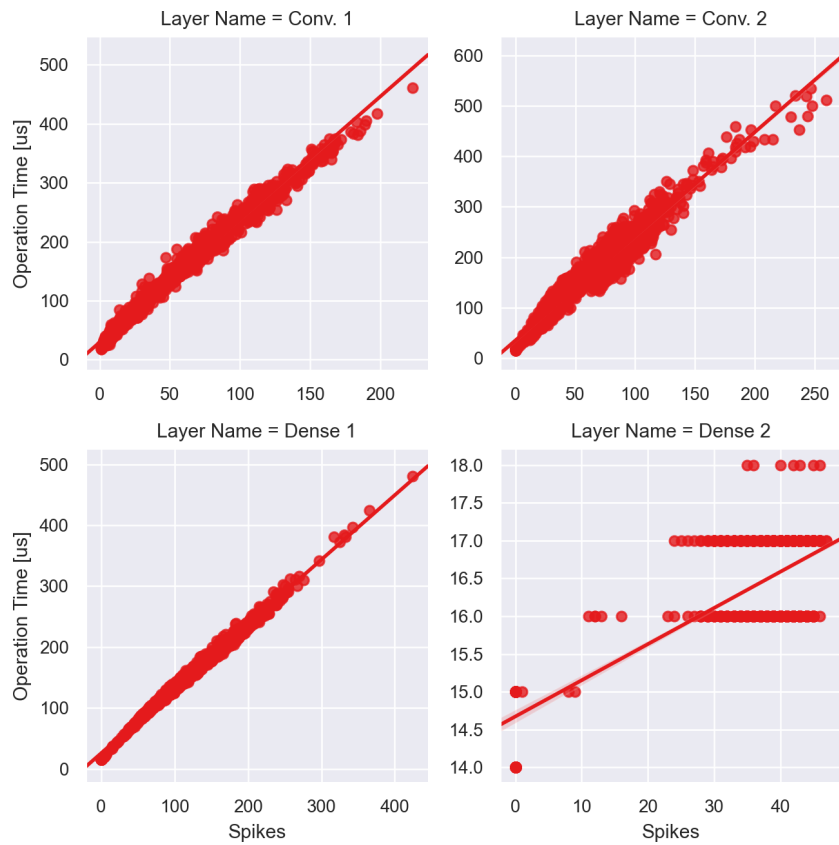


Figure 5.65: Observed characteristics of the layer operation execution time vs. the number of spikes in each layer.

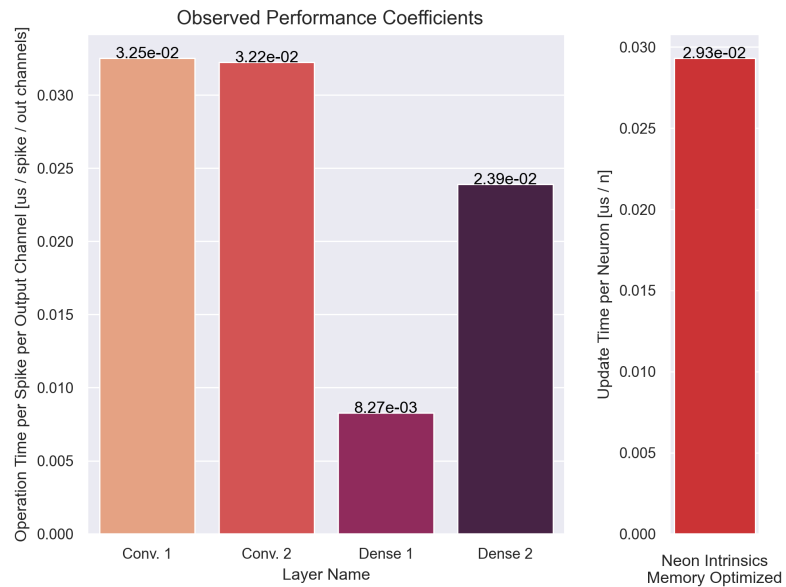


Figure 5.66: Observed performance coefficients during the model's operation.

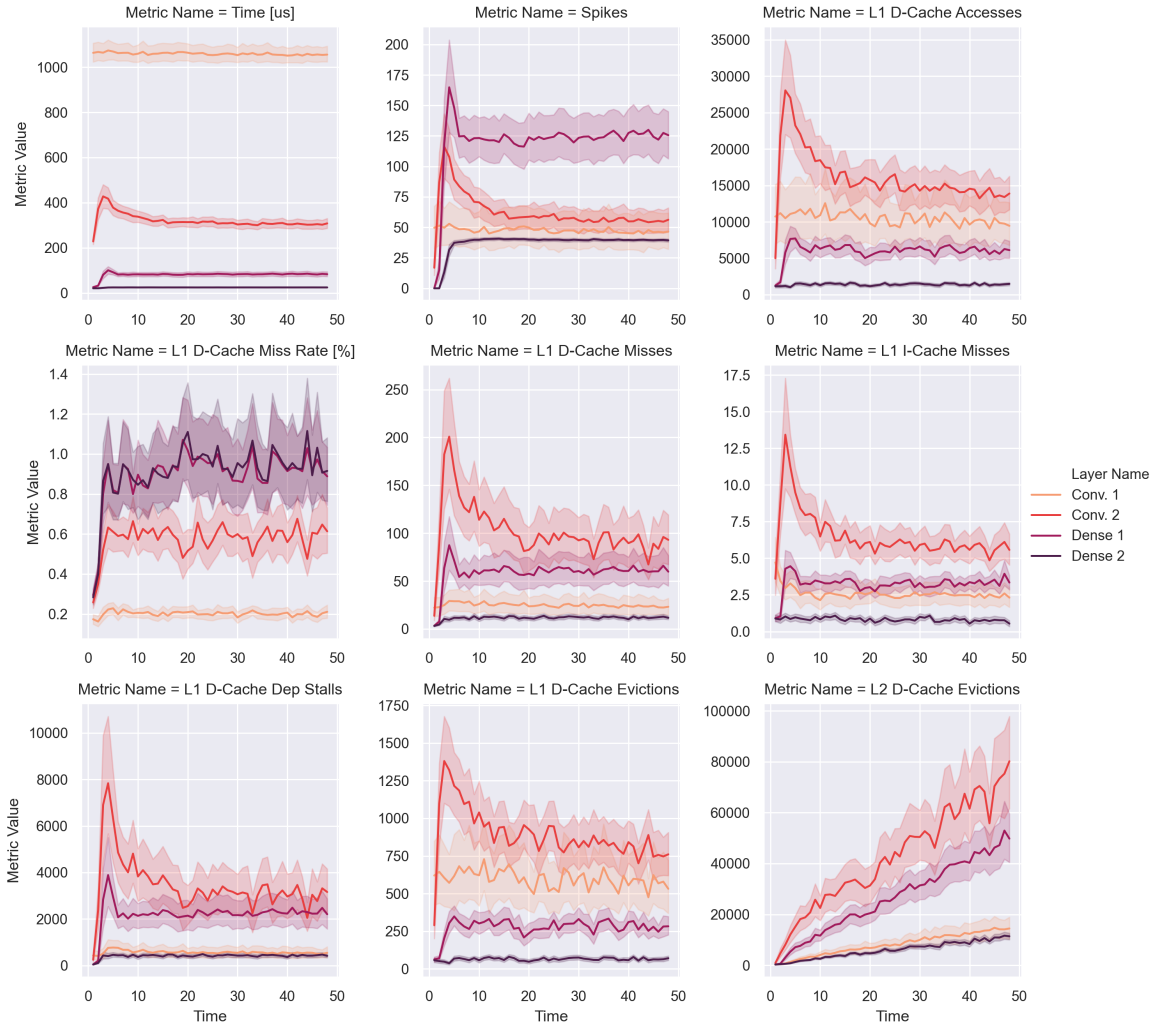


Figure 5.67: **forward** method metrics for the **F16** implementation for each time-step, averaged over 36 samples. Thick lines indicate the average and shaded areas represent the standard deviation. (Values at time 0 are removed due to a measurement error.)

5.3.3 Model timing and accuracy

In this test, the barman profiling probing functions are disabled, the execution time is still recorded and the output spikes are counted by the `Model` class in order to produce a predicted class. Since no PMU metric is recorded, the number of samples tested is only limited by the capacity of the FAT partition on the external SD/MMC card. A subset of the *1K* N-CARS test set is therefore analysed. It consists of 50 samples of each class.

The reported accuracy of the real-time Slayer algorithm on the N-CARS subset is reported in Table 5.4, along with the execution time per time-step.

Data-type	Mean [us]	Std. [us]	Min. [us]	Max. [us]	Acc. [%]	Pytorch Acc. [%]
F32	1708	118	1048	2566	81	81
F16	1640	94	1101	2858	82	81

Table 5.4: Accuracy and statistics of the execution time per time-step on the N-CARS 100 subset.

The reported time per time-step has increased compared to the profiling performed in the last section. For the most part, the subset of 100 samples contains an equal part of *Background* and *Car* classes although this was not the case in the 36-sample subset of the profiling that was mainly composed of *Background* images. Section 3.1 has shown that *Background* images are susceptible to contain less events than the other class. Therefore, the profiling therefore included a bias in the average execution time for the layer operations. Nevertheless, the main purpose of the second profiling was to validate the performance coefficients when the methods were operating on several samples.

The reported accuracy for the F32 data-type is identical to the one measured in the Pytorch Framework. Nevertheless, a modification of the output spike signals does occur and is readily visible in the example in Fig. 5.68.

Some neurons suffer from the effects of truncation of the FIR filter present in Pytorch implementation. Often, a red dot is followed by a blue dot indicating a delay in the output event. The second convolutional layer contains the largest proportion of these errors. However, the dense layers do not appear to excessively suffer from the errors of the previous layers since the number of errors diminishes and the last layer gives the correct spike signals. The fact that the accuracy of the model on the small test set of 100 samples remains unchanged further reinforce the idea that these errors do not result in a noticeable decrease in accuracy. However, this effect can be entirely removed if the Pytorch implementation is parametrized with longer FIR filters.

Concerning the F16 data-type model, the reported accuracy has been modified due to additional rounding errors. The differences between the output spike signals for each layer is shown in Fig. 5.69 for a single sample. It shows that modifications in the output spike signals occur a lot more often, and for the most part independently of the IIR truncation error. Nevertheless, the network surprisingly shows a certain robustness as the numerous errors in the second convolutional layer are attenuated in the dense layer. Finally, the output spike signals are simply shifted with respect to the Pytorch output. This effect should be further investigated, possibly using the entire dataset of N-CARS for accurate statistics.

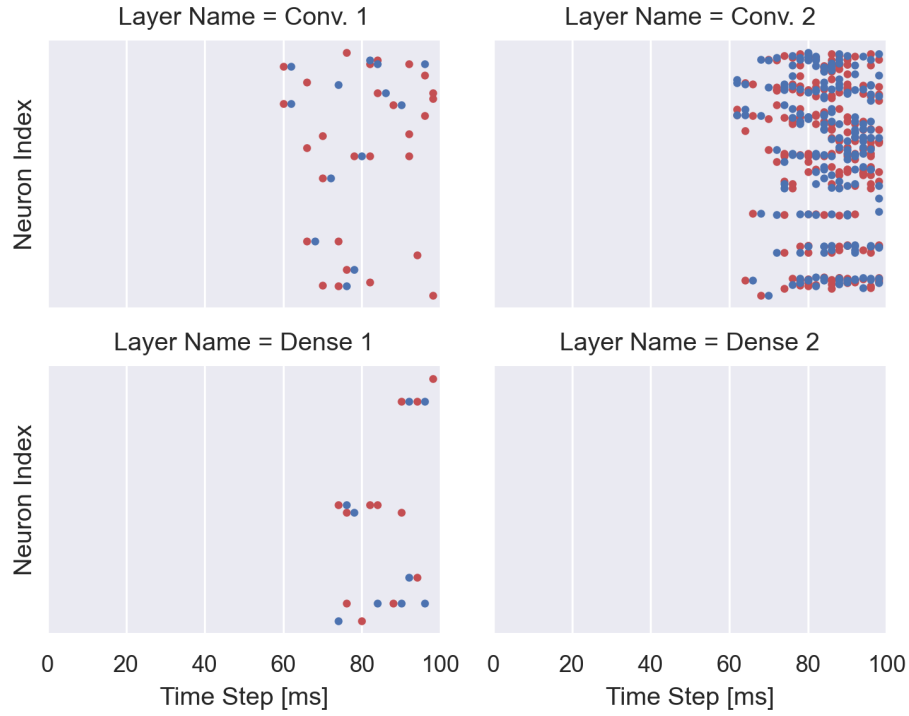


Figure 5.68: Output spike comparison of the Pytorch and C++ F32 implementations for a single sample. Red dots are events present in Pytorch in C++. Blue dots indicate the contrary.

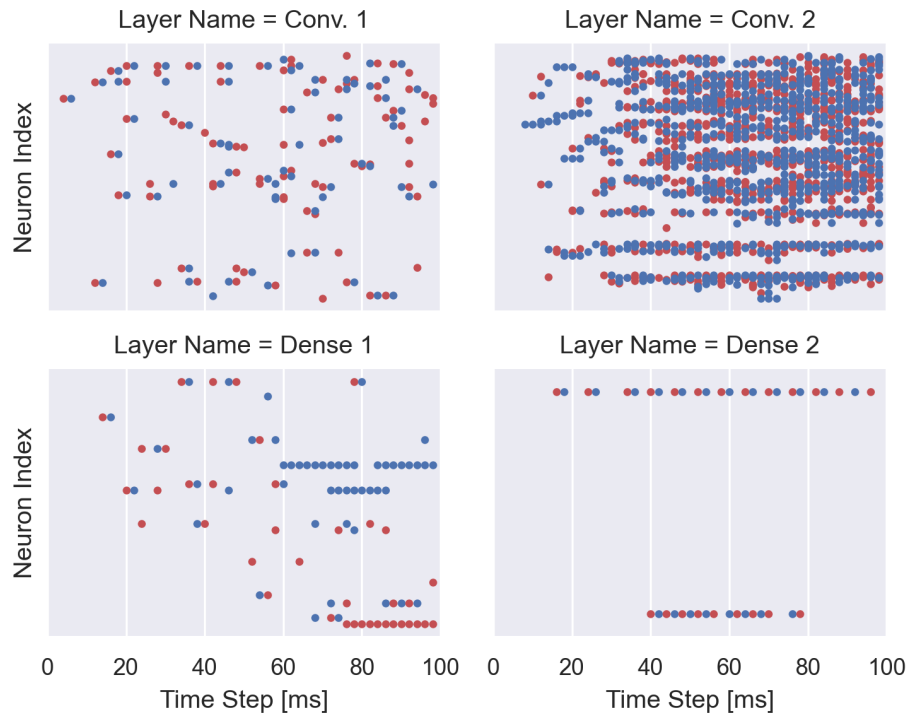


Figure 5.69: Output spike comparison of the Pytorch and C++ F16 implementations for a single sample. Red dots are events present in Pytorch in C++. Blue dots indicate the contrary.

As regards the execution time, both methods exhibit an average below the sampling period of 2000 μ s, which can be regarded as a success. However, some samples still cause a dramatic increase that exceeds the sampling period constraint. At this point two solutions are possible to solve this problem :

1. The architecture of the optimized model needs to be reconsidered in order to integrate a small number of neurons. This will decrease the **update** method execution time that takes the main proportion of the total execution time.
2. The learning process of the model can be modified to induce a smaller number of spikes that are transmitted between hidden layers. This will decrease the **layer_operation** methods execution time. Having a consistent number of spikes for each sample will also decrease the standard deviation of the execution time.

Nevertheless, the distributions of Fig. 5.55 and 5.62 for the F32 and F16 implementations indicates that the real-time constraint of 2000 μ s is satisfied pretty much all the time. Quantiles should be computed on the complete dataset to report the proportion of time-steps that respects the constraint. A gaussian estimation based on the computed average and standard deviation of the execution time in Table 5.4 indicates the constraints are respected around 95 % of the time.

The half precision floating-point implementation exhibits a slightly faster execution time and is advantageous because of its memory usage that is half the size of the single precision implementation. Moreover, it requires less accesses to the L2 cache which can greatly reduce the power consumption of the platform. This option would require further testing to measure the loss of accuracy induced by the additional rounding errors. However, results on a few samples show that the variation of accuracy remains around 1 %.

In conclusion, the optimized model mostly respects a latency of 2 ms for the F32 and F16 data-types. Accurate metrics are provided to predict the influence of the selected architecture on the execution time and could be used to explore the accuracy/latency trade-offs for optimal architectures.

5.3.4 Comparison with state-of-the-art real-time solutions

The literature generally reports the timing performances through latency measurements. However, measurement methods differ in the various reported works. There are globally 3 different ways to define latency :

- The execution time related to the processing of a single event through the entire model [19].
- The total execution time required to process a sample [77, 18].
- The time duration that separates the first event fed to the network with the output event that sets the correct output prediction [20].

Our method’s performances are therefore compared to the reported implementations of Section 2.2.5 but necessitates the following assumption. Supposing an end-to-end embedded configuration, the latency associated with the internals of the DVS and the transfer of input data from the DVS to the Cortex-A9 memory is not taken into account here. The transfer for each time-step corresponds on average to 50 AER events with a peak of 1024 events (Table 3.18). For input dimensions of $53 \times 41 \times 2$, this corresponds to a transfer of 13 bits per event and therefore an average transfer size of 162 Bytes per time-step. On the other hand, the latency of the ATIS camera is reported at $10 \mu\text{s}$ [8]. This quick computation indicates that this transfer cannot have an important impact on performances.

Sethi et al. provided an FPGA implementation of the HATS algorithm that operates on a continuous stream of events [19]. Since they reported a latency of 3.3 ms, our work offers a 40 % reduction of the latency. On the other hand, the HATS algorithm proved better performances with an accuracy 90.2 %, outperforming the 88.59 % accuracy achieved with Slayer. However, the FPGA implementation reported the use of 24-bit fixed-point data-types and a value on the loss of accuracy was not reported. Similarly, our work needs to better estimate the influence of F16 data-types on the modification of the model’s accuracy. Their power consumption measurements of 620 mW cannot yet be compared and are relegated in the discussion of future works. Early projections estimate the power consumption of the Cortex-A9 alone between 400 mW [106] and 1.8 W [107] ; however, power consumption is heavily dependent on the number of memory accesses to the L2 cache and external memory. In our situation, it also depends on the usage of the NEON extension pipelines.

Gehrig et al. proposed a cloud computing solution offering a 6.87 ms latency for the processing of an entire sample [18]. In a setup where samples would be directly provided by an interface DVS, this solution would additionally need to wait 100 ms for the complete sample to be recorded. The latency is therefore equal to 106.87 ms at minimum. Our solution distinguishes itself by assuring a continuous inference when processing the samples and only suffer a 2 ms latency. Moreover, real-time inference with the powerful ResNet model requires a GPU with a peak power estimated at 250 W which exceeds by several orders of magnitude the power consumption of the Cortex-A9 processor. On the other hand, their method achieves a 92.5 % accuracy on the N-CARS dataset.

Finally, our implementation can be compared with the work of Amir et al. on the TrueNorth platform [20]. Their implementation solves the DVS-Gesture classification task with an average latency of 104.6 ms between the first input event and the classification result and a 178 mW power budget. Since the classes in the DVS-Gesture dataset relate to different motions, most information is probably contained in the temporal dimension. As a result, the latency before detection of a movement reaches on average 104.6 ms. As stated in Section 3.4.5, our model could possibly output a correct classification result after only 30 ms thanks to the more prevalent spatial information in the N-CARS dataset.

Our implementation therefore achieves minimal latencies compared to other real-time solutions for the N-CARS dataset. Additional power measurements on our implementation are necessary in order to provide a complete and fair comparison.

The focus of this master thesis was to obtain a real-time embedded low-power implementation of a spiking neural network in an event-based computer vision application. The challenging N-CARS [17] dataset and the powerful Slayer [13] model were used in this approach. The Slayer model had already achieved state-of-the-art performances on popular neuromorphic datasets with a convolutional neural network architecture and was therefore selected in our work.

We extended the Pytorch implementation of Slayer to integrate deep learning techniques in the hope to reach maximal performances. Data augmentation methods and monitoring of performance metrics were added in order to detect and control overfitting. An optimized model with regard to classification accuracy was selected based on an hyper-parameter design space exploration, using a subset of 1000 samples of the N-CARS dataset with data augmentation. A four-step approach was chosen :

1. Some hyper-parameters were selected based on a preliminary analysis of the N-CARS dataset and Slayer model. All samples were resized to a fixed 53×41 image format using nearest interpolation. A strided convolution was selected in order to reduce the execution time of the learning phase while also reducing the complexity of the convolutional layer in the forward pass. In addition, 5×5 spatial filters were selected.
2. Hyper-parameters related to the architecture of the model were selected with a first design space exploration. An ideal configuration of two convolutional layers with 64 channels and 1 hidden dense layer with 128 neurons was found.
3. Based on optimal architectural parameters, a second design space exploration focused on selecting hyper-parameters related to the Slayer model. Among the optimal hyper-parameters, a sampling period T_s of 2 ms was selected.
4. Finally, the best model was trained on the full-size N-CARS dataset with a learning rate scheduler.

In definitive, the model achieved a **88.59 %** accuracy on the test set, rivaling with the performance of the HATS [17] model at 90.2 %. The design space exploration further showed

a great robustness against perturbations on the neuron hyper-parameters, indicating that mostly spatial information is used in the model. Moreover, an analysis of the firing rates for each layer led to two observations :

1. The average firing rate stabilizes after about 30 ms, indicating that an accurate output prediction might be obtained after that duration.
2. The features maps of the CNN show a sparsity level of at most 5 %, indicating that an event-based processing for the model operations would yield a significant reduction in execution time. Dense layers show an average sparsity of 30 %.

A real-time implementation of the optimized model was achieved in a C/C++ Baremetal software running on a single core of a Cortex-A9 processor in a Cyclone V SoC. The real-time algorithm involves a reformulation of the Slayer model that comprises the following features.

- An update rule based on an infinite impulse response filter with sampling time T_s .
- Event-based convolution and dense operations.

Multiple profilings of the baremetal software led to an optimized implementation for single and half precision floating-point data-types. Time-critical parts of the code uses 4-lane parallel SIMD operations generated with the NEON intrinsic library. Cache accesses are optimized by using aligned data-structures that are pre-fetched into the L1 data cache, therefore minimizing data misses. The update, convolution and dense methods are benchmarked in order to derive the execution time of the program in terms of the model parameters and number of spikes. The obtained **performance coefficients** indicated reliable accuracies in continuous operation with N-CARS samples.

The real-time constraint set by the sampling period of 2 ms was achieved with an average execution time estimated at **1.64** and **1.71** ms for the 16-bit and 32-bit floating-point configurations respectively. The model therefore processes events with a latency of **2 ms** and outperforms previous real-time solutions on the N-CARS task since an FPGA implementation of HATS [19] achieved a 3.3 ms latency while a cloud computing based approach with reconstructed frames [18] yielded at minimum a 106.87 ms latency.

In order to provide a fair and definitive comparison, additional efforts need to be invested in 4 elements :

1. The 16-bit floating-point implementation needs to be evaluated on the complete N-CARS dataset in order to measure the loss of accuracy compared to the Pytorch results.
2. The model accuracy with respect to the processed sample duration needs to be measured in order to provide a minimal latency of inferences.
3. A DVS camera needs to be interfaced with the Cortex-A9 processor, possibly using the programmable FPGA portion of the Cyclone V SoC. This would provide an end-to-end system that integrates the additional latencies linked to the data transfers.

4. Power measurements should be conducted to compare the design with the 620 mW power consumption achieved in HATS FPGA implementation [19].

Moreover, this work has opened a few directions for further improvements of our real-time solution in order to achieve lower latencies and possibly improve power consumption.

- The performance coefficients could ultimately be used in future works to design a model in a multi-objective optimization problem [47, 45]. Our design space exploration framework allows to integrate these metrics and could estimate a Pareto front of models with different accuracy/latency trade-offs.
- Power management capabilities of the Cortex-A9 such as standby modes and dynamic voltage/frequency scaling were not leveraged in this work. Power information can be included in the multi-objective optimization to target architectures with weights and activations that fit entirely in the cache.
- Fixed-point data-types were not considered in this work. They would potentially enable up to 16-lane parallel SIMD operations with 8-bit weights and allow a better cache usage. Alternate training techniques must be considered in order to account for the weight quantization [49].
- Finally, a dual-core implementation could be considered in order to parallelize operations in the model. The update method, accounting for 76.2% of the total execution time, has a great potential for parallelism with minimal coherency issues.

BIBLIOGRAPHY

- [1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.
- [2] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” 2014.
- [3] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, 2015.
- [4] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” 2015.
- [5] M. Verhelst and B. Moons, “Embedded deep neural network processing: Algorithmic and processor techniques bring deep learning to iot and edge devices,” *IEEE Solid-State Circuits Magazine*, vol. 9, pp. 55–65, 11 2018.
- [6] G. Gallego, T. Delbruck, G. M. Orchard, C. Bartolozzi, B. Taba, A. Censi, S. Leutenegger, A. Davison, J. Conradt, K. Daniilidis, and et al., “Event-based vision: A survey,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, p. 1–1, 2020.
- [7] C. Frenkel, M. Lefebvre, J.-D. Legat, and D. Bol, “A 0.086-mm² 12.7-pj/sop 64k-synapse 256-neuron online-learning digital spiking neuromorphic processor in 28nm cmos,” *IEEE Transactions on Biomedical Circuits and Systems*, p. 1–1, 2018.
- [8] C. Posch, D. Matolin, and R. Wohlgenannt, “A qvga 143 db dynamic range frame-free pwm image sensor with lossless pixel-level video compression and time-domain cds,” *IEEE Journal of Solid-State Circuits*, vol. 46, no. 1, pp. 259–275, 2011.
- [9] C. Brandli, R. Berner, M. Yang, S. Liu, and T. Delbruck, “A 240 × 180 130 db 3 μs latency global shutter spatiotemporal vision sensor,” *IEEE Journal of Solid-State Circuits*, vol. 49, no. 10, pp. 2333–2341, 2014.

- [10] B. Son, Y. Suh, S. Kim, H. Jung, J.-S. Kim, C.-W. Shin, K. Park, K. Lee, J. M. Park, J. Woo, Y. Roh, H. Lee, Y. M. Wang, I. A. Ovsianikov, and H. Ryu, “4.1 a 640×480 dynamic vision sensor with a $9\mu\text{m}$ pixel and 300meps address-event representation,” *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 66–67, 2017.
- [11] T. Finateu, A. Niwa, D. Matolin, K. Tsuchimoto, A. Mascheroni, E. Reynaud, P. Mostafalu, F. Brady, L. Chotard, F. LeGoff, H. Takahashi, H. Wakabayashi, Y. Oike, and C. Posch, “5.10 a 1280×720 back-illuminated stacked temporal contrast event-based vision sensor with $4.86\mu\text{m}$ pixels, 1.066geps readout, programmable event-rate controller and compressive data-formatting pipeline,” pp. 112–114, 02 2020.
- [12] B. Rueckauer, I.-A. Lungu, Y. Hu, M. Pfeiffer, and S.-C. Liu, “Conversion of continuous-valued deep networks to efficient event-driven networks for image classification,” *Frontiers in Neuroscience*, vol. 11, p. 682, 2017.
- [13] S. B. Shrestha and G. Orchard, “Slayer: Spike layer error reassignment in time,” 2018.
- [14] J. H. Lee, T. Delbruck, and M. Pfeiffer, “Training deep spiking neural networks using backpropagation,” *Frontiers in Neuroscience*, vol. 10, p. 508, 2016.
- [15] H. Markram, W. Gerstner, and P. Sjöström, “Spike-timing-dependent plasticity: A comprehensive overview,” *Frontiers in synaptic neuroscience*, vol. 4, p. 2, 07 2012.
- [16] J. M. Brader, W. Senn, and S. Fusi, “Learning real world stimuli in a neural network with spike-driven synaptic dynamics,” *Neural Computation*, vol. 19, no. 11, pp. 2881–2912, 2007.
- [17] A. Sironi, M. Brambilla, N. Bourdis, X. Lagorce, and R. Benosman, “Hats: Histograms of averaged time surfaces for robust event-based object classification,” 2018.
- [18] D. Gehrig, A. Loquercio, K. G. Derpanis, and D. Scaramuzza, “End-to-end learning of representations for asynchronous event-based data,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.
- [19] K. Sethi and M. Suri, “Optimized implementation of neuromorphic hats algorithm on fpga,” in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, 2019.
- [20] A. Amir, B. Taba, D. Berg, T. Melano, J. McKinstry, C. Di Nolfo, T. Nayak, A. Andreopoulos, G. Garreau, M. Mendoza, J. Kusnitz, M. Debole, S. Esser, T. Delbruck, M. Flickner, and D. Modha, “A low power, fully event-based gesture recognition system,” in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 7388–7397, 2017.
- [21] F. Chollet, *Deep Learning with Python*. USA: Manning Publications Co., 1st ed., 2017.
- [22] W. Hamilton, “Lecture notes in applied machine learning,” December 2019.
- [23] F. F. Rosenblatt, “Principles of neurodynamics. perceptrons and the theory of brain mechanisms,” *American Journal of Psychology*, vol. 76, p. 705, 1963.
- [24] N. Dalal and B. Triggs, “Histograms of oriented gradients for human detection,” in *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05) - Volume 1 - Volume 01*, CVPR ’05, (USA), p. 886–893, IEEE Computer Society, 2005.

- [25] D. G. Lowe, “Object recognition from local scale-invariant features,” *Proceedings of the Seventh IEEE International Conference on Computer Vision*, vol. 2, pp. 1150–1157 vol.2, 1999.
- [26] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
- [27] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.
- [28] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “ImageNet: A Large-Scale Hierarchical Image Database,” in *CVPR09*, 2009.
- [29] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” 2014.
- [30] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber, “Gradient flow in recurrent nets: the difficulty of learning long-term dependencies,” in *A Field Guide to Dynamical Recurrent Neural Networks* (Kremer and Kolen, eds.), IEEE Press, 2001.
- [31] S. Ren, K. He, R. Girshick, and J. Sun, “Faster r-cnn: Towards real-time object detection with region proposal networks,” in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS’15, (Cambridge, MA, USA), p. 91–99, MIT Press, 2015.
- [32] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 779–788, 2016.
- [33] S. Hershey, S. Chaudhuri, D. P. W. Ellis, J. F. Gemmeke, A. Jansen, C. Moore, M. Plakal, D. Platt, R. A. Saurous, B. Seybold, M. Slaney, R. Weiss, and K. Wilson, “Cnn architectures for large-scale audio classification,” in *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2017.
- [34] A. Krizhevsky, “Learning multiple layers of features from tiny images,” tech. rep., 2009.
- [35] L. Fei-Fei, R. Fergus, and P. Perona, “Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories,” *2004 Conference on Computer Vision and Pattern Recognition Workshop*, pp. 178–178, 2004.
- [36] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, “Microsoft coco: Common objects in context,” 2014.
- [37] S. Theodoridis, *Machine Learning: A Bayesian and Optimization Perspective*. USA: Academic Press, Inc., 1st ed., 2015.
- [38] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *J. Mach. Learn. Res.*, vol. 15, p. 1929–1958, Jan. 2014.

- [39] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.
- [40] NVIDIA Corporation, “NVIDIA CUDA C++ programming guide,” 2020. Version PG-02829-001 v11.0.
- [41] W. Falcon, “Pytorch lightning,” *GitHub*. *Note: <https://github.com/PyTorchLightning/pytorch-lightning>* Cited by, vol. 3, 2019.
- [42] L. Biewald, “Experiment tracking with weights and biases,” 2020. Software available from wandb.com.
- [43] A. Buslaev, “Albumentations: fast and flexible image augmentations,” *ArXiv e-prints*, 2018.
- [44] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, “Optuna: A next-generation hyperparameter optimization framework,” in *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.
- [45] C. Blum and A. Roli, “Metaheuristics in combinatorial optimization: Overview and conceptual comparison,” *ACM Comput. Surv.*, vol. 35, p. 268–308, Sept. 2003.
- [46] Y.-H. Chen, J. Emer, and V. Sze, “Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks,” in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, p. 367–379, IEEE Press, 2016.
- [47] S. Smithson, G. Yang, W. Gross, and B. Meyer, “Neural networks designing neural networks: Multi-objective hyper-parameter optimization,” pp. 1–8, 11 2016.
- [48] A. Lavin and S. Gray, “Fast algorithms for convolutional neural networks,” 2015.
- [49] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, “Fixed point quantization of deep convolutional networks,” 2015.
- [50] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, “Xnor-net: Imagenet classification using binary convolutional neural networks,” 2016.
- [51] K. Goetschalckx, B. Moons, S. Lauwereins, M. Andraud, and M. Verhelst, “Optimized hierarchical cascaded processing,” *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, no. 4, pp. 884–894, 2018.
- [52] B. Moons and M. Verhelst, “A 0.3-2.6 tops/w precision-scalable processor for real-time large-scale convnets,” 2016.
- [53] J. Xue, J. Li, and Y. Gong, “Restructuring of deep neural network acoustic models with singular value decomposition,” *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*, pp. 2365–2369, 01 2013.

- [54] S. Han, J. Pool, J. Tran, and W. J. Dally, “Learning both weights and connections for efficient neural networks,” 2015.
- [55] X. Liu, “Pruning of winograd and fft based convolution algorithm,” 2016.
- [56] L. Camunas-Mesa, C. Zamarreno-Ramos, A. Linares-Barranco, A. J. Acosta-Jimenez, T. Serrano-Gotarredona, and B. Linares-Barranco, “An event-driven multi-kernel convolution processor module for event-driven vision sensors,” *IEEE Journal of Solid-State Circuits*, vol. 47, no. 2, pp. 504–517, 2012.
- [57] G. Orchard, A. Jayawant, G. K. Cohen, and N. Thakor, “Converting static image datasets to spiking neuromorphic datasets using saccades,” *Frontiers in Neuroscience*, vol. 9, p. 437, 2015.
- [58] G. Orchard, A. Jayawant, G. K. Cohen, and N. Thakor, “Converting static image datasets to spiking neuromorphic datasets using saccades,” *Frontiers in Neuroscience*, vol. 9, p. 437, 2015.
- [59] T. Masquelier and S. J. Thorpe, “Unsupervised learning of visual features through spike timing dependent plasticity,” *PLOS Computational Biology*, vol. 3, pp. 1–11, 02 2007.
- [60] P. O’Connor, D. Neil, S.-C. Liu, T. Delbruck, and M. Pfeiffer, “Real-time classification and sensor fusion with a spiking deep belief network,” *Frontiers in Neuroscience*, vol. 7, p. 178, 2013.
- [61] P. Lichtsteiner, C. Posch, and T. Delbruck, “A 128 128 120 db 15 s latency asynchronous temporal contrast vision sensor,” 2006.
- [62] T. S. Lande, J. . Marienborg, and Y. Berg, “Neuromorphic cochlea implants,” in *2000 IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 4, pp. 401–404 vol.4, 2000.
- [63] D. M. J. Spencer, “Slides lecture 3 - synapses in neural information processing,” 2018.
- [64] R. Wang and A. van Schaik, “Breaking liebigs law: An advanced multipurpose neuromorphic engine,” *Frontiers in Neuroscience*, vol. 12, p. 593, 2018.
- [65] T. Pirson, “Training ultra-low-power spiking neural networks for neuromorphic IoT vision sensing and recognition,” Master’s thesis, Ecole Polytechnique de Louvain, Université Catholique de Louvain, 2019. Prom. Bol, David ; Frenkel, Charlotte.
- [66] W. Gerstner, “Time structure of the activity in neural network models,” *Phys. Rev. E*, vol. 51, pp. 738–758, Jan 1995.
- [67] C. Frenkel, *Bottom-up and top-down neuromorphic processor design: Unveiling roads to embedded cognition*. PhD thesis, 01 2020.
- [68] E. Stomatias, M. Soto, T. Serrano-Gotarredona, and B. Linares-Barranco, “An event-driven classifier for spiking neural networks fed with synthetic or dynamic vision sensor data,” *Frontiers in Neuroscience*, vol. 11, p. 350, 2017.
- [69] L. R. Iyer, Y. Chua, and H. Li, “Is neuromorphic mnist neuromorphic? analyzing the discriminative power of neuromorphic datasets in the time domain,” 2018.

- [70] X. Lagorce, G. Orchard, F. Galluppi, B. Shi, and R. Benosman, “Hots: A hierarchy of event-based time-surfaces for pattern recognition,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, pp. 1346–1359, 2017.
- [71] A. C. Bovik, M. Clark, and W. S. Geisler, “Multichannel texture analysis using localized spatial filters,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 1, pp. 55–73, 1990.
- [72] P. U. Diehl, B. U. Pedroni, A. Cassidy, P. Merolla, E. Neftci, and G. Zarrella, “True-happiness: Neuromorphic emotion recognition on truenorth,” in *2016 International Joint Conference on Neural Networks (IJCNN)*, pp. 4278–4285, 2016.
- [73] A. S. Cassidy, J. Sawada, P. Merolla, J. V. Arthur, R. Alvarez-Icaza, F. Akopyan, B. L. Jackson, and D. S. Modha, “Truenorth: A high-performance, low-power neurosynaptic processor for multi-sensory perception, action, and cognition,” 2016.
- [74] W. Zhang and P. Li, “Spike-train level backpropagation for training deep recurrent spiking neural networks,” 2019.
- [75] S. Bohte, J. Kok, and J. Poutré, “Spikeprop: backpropagation for networks of spiking neurons,” vol. 48, pp. 419–424, 01 2000.
- [76] PROPHESEE, Metavision for Machines.
- [77] H. Rebecq, R. Ranftl, V. Koltun, and D. Scaramuzza, “Events-to-video: Bringing modern computer vision to event cameras,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.
- [78] B. V. Benjamin, P. Gao, E. McQuinn, S. Choudhary, A. R. Chandrasekaran, J. Bussat, R. Alvarez-Icaza, J. V. Arthur, P. A. Merolla, and K. Boahen, “Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations,” *Proceedings of the IEEE*, vol. 102, no. 5, pp. 699–716, 2014.
- [79] A. Grübl, S. Billaudelle, B. Cramer, V. Karasenko, and J. Schemmel, “Verification and design methods for the brainscales neuromorphic hardware system,” 2020.
- [80] N. Qiao, H. Mostafa, F. Corradi, M. Osswald, F. Stefanini, D. Sumislawska, and G. Indiveri, “A reconfigurable on-line learning spiking neuromorphic processor comprising 256 neurons and 128k synapses,” *Frontiers in Neuroscience*, vol. 9, p. 141, 2015.
- [81] S. Moradi, N. Qiao, F. Stefanini, and G. Indiveri, “A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (dynaps),” *IEEE Transactions on Biomedical Circuits and Systems*, vol. 12, p. 106–122, Feb 2018.
- [82] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, P. Joshi, A. Lines, A. Wild, and H. Wang, “Loihi: A neuromorphic manycore processor with on-chip learning,” *IEEE Micro*, vol. PP, pp. 1–1, 01 2018.
- [83] C. Frenkel, J.-D. Legat, and D. Bol, “Morphic: A 65-nm 738k-synapse/mm² quad-core binary-weight digital neuromorphic processor with stochastic spike-driven online learning,” *IEEE Transactions on Biomedical Circuits and Systems*, vol. 13, p. 999–1010, Oct 2019.

- [84] S. Furber, F. Galluppi, S. Temple, and L. Plana, “The spinnaker project,” *IEEE Proceedings*, vol. 102, no. 5, pp. 652–665, 2014.
- [85] H. Bourlès, *Fundamentals of Advanced Mathematics 2*. 02 2018.
- [86] R. Jolivet, T. J., and W. Gerstner, “The spike response model: A framework to predict neuronal spike trains,” in *Artificial Neural Networks and Neural Information Processing — ICANN/ICONIP 2003* (O. Kaynak, E. Alpaydin, E. Oja, and L. Xu, eds.), (Berlin, Heidelberg), pp. 846–853, Springer Berlin Heidelberg, 2003.
- [87] S. J. Reddi, S. Kale, and S. Kumar, “On the convergence of adam and beyond,” 2019.
- [88] ARM, “Arm cortex-a series programmer’s guide,” 2013. Version 4.0.
- [89] ARM, “Neon programmer’s guide,” 2013. Version 1.0.
- [90] ARM, “Arm architecture reference manual, armv7-a and armv7-r edition,” 2018.
- [91] ARM, “Arm ds-5 streamline user guide,” 2016. Version 6.1.
- [92] ARM, “Corelink level 2 cache controller l2c-310,” 2012. Revision r3p3.
- [93] ARM, “Cortex-a9 technical reference manual,” 2012. Revision r4p1.
- [94] ARM, “Arm cortex-a9 mpcore technical reference manual,” 2016. Revision r4p1.
- [95] ARM, “Cortex-a9 neon media processing engine technical reference manual,” 2011. Revision r3p0.
- [96] Richard M. Stallman and the GCC Developer Community, “Using the gnu compiler collection,” 2016. Revision r3p0.
- [97] S. L. H. . D. M. Harris, *Digital Design and Computer Architecture, ARM Edition*. Morgan Kaufmann, 2016.
- [98] Terasic, “De10-nano user manual,” 2019.
- [99] “Ieee standard for floating-point arithmetic,” *IEEE Std 754-2008*, pp. 1–70, 2008.
- [100] ARM, “Procedure call standard for the arm® architecture,” 2009. ARM IHI 0042D.
- [101] Altera Corporation, “Bare metal user guide,” 2015. Version ug-01665.
- [102] Mentor, a Siemens Business, “Sourcery codebench lite arm altera eabi getting started,” 2016.
- [103] J. O. Smith, *Introduction to Digital Filters with Audio Applications*. <http://www.w3k.org/books/>: W3K Publishing, 2007.
- [104] J. Yopez and S. Ko, “Stride 2 1-d, 2-d, and 3-d winograd for convolutional neural networks,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 4, pp. 853–863, 2020.
- [105] J. van der Zijp, “Fast half float conversions,” 2010.
- [106] C. Walls, *Embedded Software, Second Edition: The Works*. USA: Newnes, 2nd ed., 2012.
- [107] Intel, “Cyclone v fpga products,” 2020.

Appendices

APPENDIX A

PYTORCH YAML CONFIGURATION FILES

A.1 Baseline model

```
1 pytorch:
2   architecture:
3     layer_0:
4       name: 'conv',
5       params: { inChannels: 2, outChannels: 16, kernelSize: 5, stride: 2 }
6     layer_1:
7       name: 'conv',
8       params: { inChannels: 16, outChannels: 16, kernelSize: 5, stride: 2 }
9     layer_2:
10      name: 'dense',
11      params: { inFeatures: [11, 8, 16], outFeatures: 2 }
12   dataset:
13     batch: 32
14     size: {height: 41, width: 53}
15     train/val: 0.8
16     path: { in: NCARS/, test: NCARS/test.txt, train: NCARS/train.txt}
17   training:
18     early stopping: { patience: 6 }
19     epoch: 50
20     optimizer:
21       name: 'Adam'
22       params: { lr: 1.0e-2, amsgrad: True }
23     scheduler: None
24   slayer:
25     error:
26       type: numSpikes
27       tgtSpikeRegion: {start: 0, stop: 100}
28       tgtSpikeCount: {true: 90, false: 10}
29     neuron:
30       {
31         scaleRef: 2
32         scaleRho: 1
33         tauRef: 1
34         tauRho: 1
35         tauSr: 10
36         theta: 10
37         type: SRMALPHA
38       }
39   simulation: {Ts: 1, tSample: 100}
```

Figure A.1: YAML Configuration file for the Baseline Slayer Model.

A.2 Optimized model

```
1  pytorch:
2    architecture:
3      layer_0:
4        name: conv
5        params: { inChannels: 2, kernelSize: 5, outChannels: 64, stride: 2 }
6      layer_1:
7        name: conv
8        params: { inChannels: 64, kernelSize: 5, outChannels: 64, stride: 2 }
9      layer_2:
10       name: dense
11       params:
12         inFeatures: [11, 8, 64]
13         outFeatures: 128
14      layer_3:
15       name: dense
16       params: { inFeatures: 128, outFeatures: 2 }
17    dataset:
18      batch: 32
19      path: { in: NCARS/, test: NCARS/test.txt, train: NCARS/train.txt }
20      size: { height: 41, width: 53 }
21      train/val: 0.8
22    training:
23      early stopping: { patience: 10 }
24      epoch: 50
25      optimizer:
26        name: Adam
27        params: { amsgrad: true, lr: 0.07434984021927094 }
28      scheduler:
29        name: MultiStepLR
30        params: { milestones: [15, 25], gamma: 0.1 }
31  slayer:
32    error:
33      tgtSpikeCount: { false: 7, true: 27 }
34      tgtSpikeRegion: { start: 0, stop: 100 }
35      type: numSpikes
36    neuron:
37      {
38        scaleRef: 4.827982821042214,
39        scaleRho: 1,
40        tauRef: 3.067269828600086,
41        tauRho: 8.87348147727414,
42        tauSr: 7.361087683982831,
43        theta: 11.103819921413564,
44        type: SRMALPHA,
45      }
46    simulation: { Ts: 2, tSample: 100 }
```

Figure A.2: YAML Configuration file for the Optimized Slayer Model.

B.1 Virtual address space

One of the main functions of the MMU is to provide the Cortex-A9 with a *virtual memory space*. Unlike the *physical memory space* that connects memory devices and other peripherals to fixed addresses, a virtual memory space is a layer that allows to choose a mapping between virtual addresses and physical addresses. This has the following advantages :

- It allows to use the same memory addresses for programs running concurrently on the processor.
- A contiguous memory space can be created even if the physical memory space is fragmented (Fig. 4.13).
- Additional information about memory regions can be stored in the mapping table.

The role of the MMU is to perform address translations at runtime between virtual addresses used in the program and physical addresses used by the hardware using *translation tables*. A translation table is an array in memory, sorted by virtual address, where each entry maps to a physical address and *memory attributes*.

B.2 Translation tables

The Cortex-A9 uses a translation process that involves two levels of translation tables that are called the L1 and L2 translation tables [88]. The L1 translation table partitions the 4 GB of virtual memory space into 4096 32-bit wide entries. Each entry is therefore responsible for a 1 MB region of memory and either contains

- a descriptor for a 1 MB section that provides the physical address and memory attributes. The descriptor might be part of a 16 MB super-section, whose purpose is explained in Section B.3.
- a memory address forwarding to an L2 translation table, further describing the memory region into *pages*.
- a fault entry indicating that the region is not mapped.

The L1 translation table lies in memory and its base address is stored in the *Translation Table Base Register* (TTBR) of the MMU. It occupies 16 KB in total and must be aligned in memory to a 16 KB boundary. The process by which a 1 MB descriptor is translated into a physical address is illustrated in Fig. B.1.

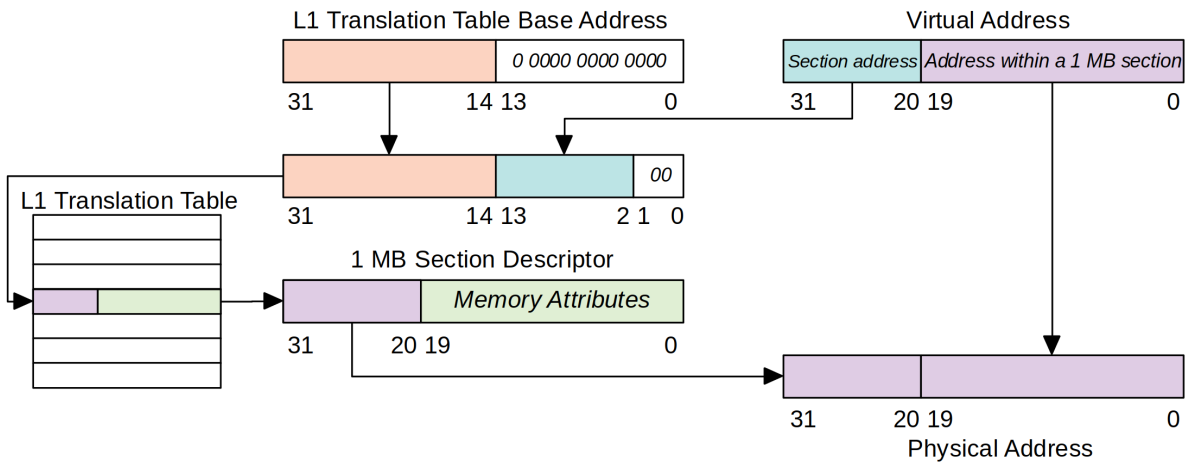


Figure B.1: Virtual address translation of a 1 MB section descriptor, inspired from [88].

Since the L1 translation table is aligned to a 16 KB boundary in memory, only `bit[31:14]` of the TTBR are necessary to describe its address. As for the virtual address, it is mapped to a descriptor for a 1 MB section aligned memory thus only `bit[31:20]` are needed to find the entry in the translation table. The fetched descriptor is a 32-bit word containing the address of the physical 1 MB section and associated memory attributes. Therefore, the physical address is formed by joining `bit[31:20]` of the descriptor with the lower bits of the virtual address. The memory attributes include the following information :

- The *memory cacheability* indicates if the memory content can be stored in the caches.
- The *memory shareability* indicates if the memory can be accessed by more than one core.
- The *memory order* indicates if the memory accesses can be buffered (Section 4.4.4).
- The *access permission* indicates processor modes that are authorized to access the memory.
- The *inner and outer cache policy* dictates the policy of L1 and L2 caches (Section 4.4.3).

An L2 translation table exists for each corresponding entry in the L1 translation table. It partitions the 1 MB section into 256 32-bit wide entries. Each entry is therefore responsible for a 4 KB region and either contains

- a descriptor for a 4 KB page that provides the physical address and memory attributes. The descriptor might be part of a 64 KB large page.
- a fault entry indicating that the region is not mapped.

Each L2 translation table occupies 1 KB in memory as must be aligned to a 1 KB boundary. The process that translates a 4 KB descriptor to a physical address is nearly identical to the one for the 1 MB section as is illustrated in Fig. B.2.

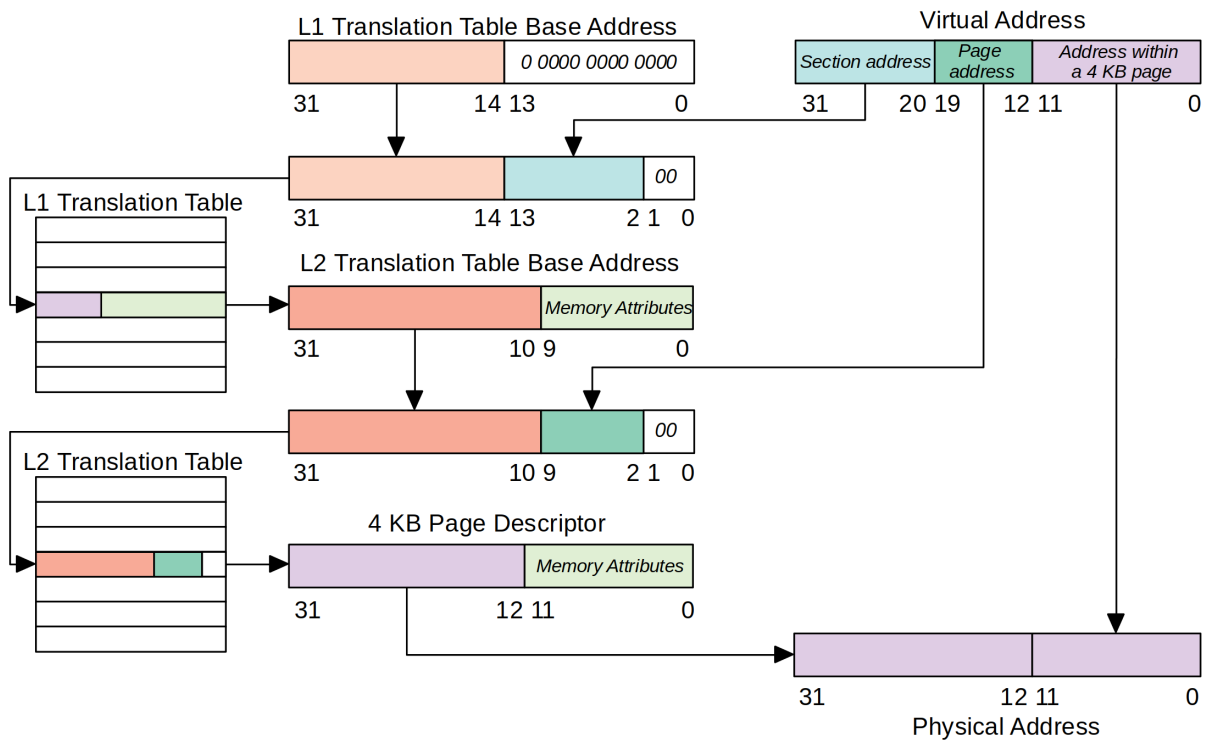


Figure B.2: Virtual address translation of a 4 KB page descriptor, inspired from [88].

Since the L2 translation table is aligned to a 1 KB boundary in memory, only **bit**[31:10] of L1 descriptor are necessary to describe its address. As for the virtual address, it is mapped to a descriptor for a 4 KB page aligned memory thus only **bit**[19:12] are needed to find the entry in the translation table. The fetched descriptor is a 32-bit word containing the address of the physical 4 KB page and associated memory attributes. Therefore, the physical address is formed by joining **bit**[31:20] of the L1 descriptor, **bit**[19:12] of the L2 descriptor and the remaining lower bits of the virtual address.

B.3 Table lookaside buffer

The *Translation lookaside buffer* (TLB) is a cache that stores recently used L1 and L2 translation table entries. When an address translation is requested, the MMU first checks if

the requested entry of the translation table exists inside the TLB before fetching it from main memory. If the TLB contains the entry, there is a *TLB hit* otherwise there is a *TLB miss*. Additionally, the Cortex-A9 is equipped with 2 micro-TLB (μ TLB) each associated with the L1 instruction and data cache. As in other hierarchical cache systems, descriptors are first searched in the μ TLB and then in the *Main TLB* if it is not available.

The μ TLB of the Cortex-A9 are built as fully-associative caches with 32 entries that can perform a lookup of the virtual address in a single cycle [94]. The Main TLB is a 2-way associative cache with 128 entries that requires several cycles necessary to lookup a virtual address.

The choice of section's and page's sizes introduces the following considerations regarding the TLBs. Small 4 KB pages provide a narrow control on the memory attributes over small regions but require more translation table entries. There is therefore a higher probability of misses in the TLBs. Sections of 1 MB on the other hand only require L1 translation table entries and lead to better TLB performances. Finally, 16 MB super-sections and 64 KB large pages occupy a single TLB entry for multiple translation table entries.

APPENDIX C

C++ AND ASSEMBLY METHODS

C.1 Update method

C.1.1 32-bit floating-point implementations

C.1.1.1 Vectorizing compiler approach

```
1 unsigned int update_state(float** state, const float* filter, unsigned int* output, const←
   unsigned int len)
2 {
3     // Number of output spikes
4     unsigned int len_output = 0u;
5
6     float* si{state[0]};
7     float* s1{state[1]}; // eps and nu states are interleaved
8     float* s2{state[2]};
9     unsigned int n;
10    for(n = 0;
11        n < len;
12        ++n, si += 1, s1 += 2, s2 += 2)
13    {
14        bool spike = ((filter[0] * s1[0] + filter[1] * s1[1]) >= filter[6]);
15        if (spike)
16            output[len_output++] = n;
17
18        s2[0] = si[0] + filter[2] * s1[0] + filter[3] * s2[0]; // eps iir
19        s2[1] = ((float) spike) + filter[4] * s1[1] + filter[5] * s2[1]; // nu iir
20        si[0] = 0.0f;
21    }
22
23    // Update states s2 <-> s1 (change pointers head)
24    float* temp = state[1];
25    state[1] = state[2];
26    state[2] = temp;
27
28    return len_output;
29 }
```

Figure C.1: update method for the C implementation.

```

1 00000000 <update_state_c>:
2   0:   e3d33003    bics    r3, r3, #3
3   4:   e92d47f0    push    {r4, r5, r6, r7, r8, r9, sl, lr}
4   8:   e1a08000    mov     r8, r0
5   c:   e8900640    ldm     r0, {r6, r9, sl}
6  10:   0a00002a    beq     c0 <update_state_c+0xc0>
7  14:   e3a05000    mov     r5, #0
8  18:   e3a07000    mov     r7, #0
9  1c:   e1a00005    mov     r0, r5
10 20:   e1a04009    mov     r4, r9
11 24:   e1a0e00a    mov     lr, sl
12 28:   e28ac004    add     ip, sl, #4
13 2c:   ed946a01    vldr    s12, [r4, #4]
14 30:   e28cc008    add     ip, ip, #8
15 34:   eed17a01    vldr    s15, [r1, #4]
16 38:   e2844008    add     r4, r4, #8
17 3c:   ed545a02    vldr    s11, [r4, #-8]
18 40:   e28ee008    add     lr, lr, #8
19 44:   eed16a00    vldr    s13, [r1]
20 48:   ed917a06    vldr    s14, [r1, #24]
21 4c:   ee667a27    vmul.f32 s15, s12, s15
22 50:   eed14a02    vldr    s9, [r1, #8]
23 54:   ed915a04    vldr    s10, [r1, #16]
24 58:   ee457aa6    vmla.f32 s15, s11, s13
25 5c:   eef47ac7    vcmpe.f32 s15, s14
26 60:   eef1fa10    vmrs    APSR_nzcv, fpscr
27 64:   eef06a67    vmov.f32 s13, s15
28 68:   a7825100    strge   r5, [r2, r0, lsl #2]
29 6c:   e2855001    add     r5, r5, #1
30 70:   edd67a00    vldr    s15, [r6]
31 74:   a2800001    addge   r0, r0, #1
32 78:   e4867004    str     r7, [r6], #4
33 7c:   e1550003    cmp     r5, r3
34 80:   ed1c4a03    vldr    s8, [ip, #-12]
35 84:   ed917a03    vldr    s14, [r1, #12]
36 88:   ee277a04    vmul.f32 s14, s14, s8
37 8c:   ee057aa4    vmla.f32 s14, s11, s9
38 90:   ee377a27    vadd.f32 s14, s14, s15
39 94:   ed0c7a03    vstr    s14, [ip, #-12]
40 98:   ed1e7a01    vldr    s14, [lr, #-4]
41 9c:   eed17a05    vldr    s15, [r1, #20]
42 a0:   ee677a87    vmul.f32 s15, s15, s14
43 a4:   ee467a05    vmla.f32 s15, s12, s10
44 a8:   ee777aa6    vadd.f32 s15, s15, s13
45 ac:   ed4e7a01    vstr    s15, [lr, #-4]
46 b0:   1affffdd    bne     2c <update_state_c+0x2c>
47 b4:   e588a004    str     s1, [r8, #4]
48 b8:   e5889008    str     r9, [r8, #8]
49 bc:   e8bd87f0    pop     {r4, r5, r6, r7, r8, r9, sl, pc}
50 c0:   e1a00003    mov     r0, r3
51 c4:   eaaffffa    b       b4 <update_state_c+0xb4>

```

Figure C.2: update method Assembly code for the C implementation.

C.1.1.2 NEON intrinsics approach

```

1 #include <arm_neon.h>
2
3 #ifdef PREFETCH
4 #define PLD(x)    asm ("pld [%0]" :: "r"(x))
5 #else
6 #define PLD(x)
7 #endif
8
9 #ifdef ALIGNED
10 #define ASSUME_ALIGNED(type, ptr) reinterpret_cast<type>(__builtin_assume_aligned(←
    reinterpret_cast<void*>(ptr), 32))
11 #else

```

```

12 #define ASSUME_ALIGNED(type, ptr) ptr
13 #endif
14
15 unsigned int update_state(float** __restrict state, const float* __restrict filter, ←
    unsigned int* output, const unsigned int len) // Restrict says pointers are not ←
    overlapping
16 {
17     // Common registers between iterations
18
19     // Number of output spikes
20     unsigned int len_output = 0u;
21     // Load filter coeffs + threshold
22
23     // 63      32 31      0 : D register
24     // [forw1_nu forw1_eps]
25     float32x2_t vforw_duo = vld1_f32(filter);
26
27     // 63      32 31      0 : D register
28     // [back2_eps back1_eps]
29     float32x2_t vback_eps = vld1_f32(filter + 2);
30
31     // 63      32 31      0 : D register
32     // [back2_nu back1_nu ]
33     float32x2_t vback_nu  = vld1_f32(filter + 4);
34
35     // 127 96 95 64 63 32 31  0 : Q register
36     // [theta theta theta theta]
37     float32x4_t vtheta = vld1q_dup_f32(filter + 6);
38
39     // 63      32 31      0 : D register
40     // [1.0f  1.0f  1.0f  1.0f ]
41     float32x4_t vones  = vmovq_n_f32(1.0f);
42
43     // 63      32 31      0 : D register
44     // [0.0f  0.0f  0.0f  0.0f ]
45     float32x4_t vzeros = vmovq_n_f32(0.0f);
46
47     // Buffer array for the bit-mask allocated on the stack
48     uint32_t temp_output[4];
49
50     // Alias states
51     // input is loaded in a cache line but uses only 4 entries in the current iteration
52     float32_t* si{ASSUME_ALIGNED(float32_t*, state[0])};
53
54     // eps and nu are interleaved to use a cache line (8 words = 4 eps + 4 nu) (Optimal ←
        for 4 // SIMD)
55     float32_t* s1{ASSUME_ALIGNED(float32_t*, state[1])};
56     float32_t* s2{ASSUME_ALIGNED(float32_t*, state[2])};
57
58     unsigned int n;
59 #if (UPDATE_DESIGN == 1) // State separated implementation
60     for(n = 0;
61         n < len;
62         n += 4, si += 4, s1 += 8, s2 += 8) // Number of states must be a multiple of 4
63     {
64         PLD(s1 + 8);
65         PLD(s2 + 8);
66         PLD(si + 4);
67 #elif (UPDATE_DESIGN == 2) // State interleaved implementation
68     for(n = 0;
69         n < len;
70         n += 4, si += 4, s1 += 16, s2 += 16) // Number of states must be a multiple of 4
71     {
72         PLD(s1 + 32);
73         PLD(s2 + 32);
74         PLD(si + 8);
75 #endif
76     // Load states & input : avoid load stalls (but compilation normally reschedule ←
        if necessary)
77     float32x4x2_t vs1  = vld2q_f32(s1); // Deinterleave (eps | nu | eps | nu ...)
78     float32x4x2_t vs2  = vld2q_f32(s2);
79     float32x4_t vsinput = vld1q_f32(si);
80     vst1q_f32(si, vzeros); // Clear input (while still cached)
81

```

```

82 // Forward filter equation : u = forw1_eps * s1_eps + forw1_nu * s1_nu
83 // spike = (u >= theta)
84
85 // temp = f_1,eps * s_1,eps
86 float32x4_t vspike = vmulq_lane_f32(vs1.val[0], vforw_duo, 0);
87
88 // u = temp + f_1,nu * s_1,nu
89 vspike = vmlaq_lane_f32(vspike, vs1.val[1], vforw_duo, 1);
90
91 // mask = (u >= theta) (all 1 if true, all 0 otherwise)
92 uint32x4_t vcond = vcgeq_f32(vspike, vtheta);
93
94 // spike = 1.0 if mask all 1, 0.0 otherwise
95 vspike = vbslq_f32(vcond, vones, vzeros);
96
97 // Output spikes store
98 vst1q_u32(temp_output, vcond);
99 for (unsigned int m = 0; m < 4; ++m)
100     if (temp_output[m])
101         output[len_output++] = (n + m);
102
103 // Backward filter equation : s0_eps = input + back1_eps * s1_eps + back2_eps * ←
104 // s2_eps s0_nu = spike + back1_nu * s1_nu + back2_nu * ←
105 // s2_nu
106 float32x4x2_t vs0;
107
108 // Eps filter equation : temp_eps = input + back1_eps * s1_eps
109 vs0.val[0] = vmlaq_lane_f32(vsi, vs1.val[0], vback_eps, 0);
110
111 // Nu filter equation : temp_nu = spike + back1_nu * s1_nu
112 vs0.val[1] = vmlaq_lane_f32(vspike, vs1.val[1], vback_nu, 0);
113
114 // Eps filter equation : s0_eps = temp_eps + back2_eps * s2_eps
115 vs0.val[0] = vmlaq_lane_f32(vs0.val[0], vs2.val[0], vback_eps, 1);
116
117 // Nu filter equation : s0_nu = temp_nu + back2_nu * s2_nu
118 vs0.val[1] = vmlaq_lane_f32(vs0.val[1], vs2.val[1], vback_nu, 1);
119
120 // Store result in the s2 state memory
121 vst2q_f32(s2, vs0); // Interleave (eps | nu | eps | nu ...)
122 }
123
124 // Update states s2 ← s1 (change pointers head)
125 float32_t* temp = state[1];
126 state[1] = state[2];
127 state[2] = temp;
128
129 return len_output;
130 }

```

Figure C.3: update method for the F32 NEON implementation.

```

1 00000000 <update_state_f32>:
2 0: e92d4ff0 push {r4, r5, r6, r7, r8, r9, s1, fp, lr}
3 4: e281c018 add ip, r1, #24
4 8: e281e008 add lr, r1, #8
5 c: e2814010 add r4, r1, #16
6 10: e3530000 cmp r3, #0
7 14: e1a08000 mov r8, r0
8 18: ed2d8b04 vpush {d8-d9}
9 1c: e24dd014 sub sp, sp, #20
10 20: f42e678f vld1.32 {d6}, [lr]
11 24: e590e000 ldr lr, [r0]
12 28: f424578f vld1.32 {d5}, [r4]
13 2c: e5904004 ldr r4, [r0, #4]
14 30: f4ececfa vld1.32 {d30[]-d31[]}, [ip]
15 34: e590c008 ldr ip, [r0, #8]
16 38: f421778f vld1.32 {d7}, [r1]
17 3c: 0a000035 beq 118 <update_state_f32+0x118>
18 40: e3a01000 mov r1, #0

```

```

19 44: f2872f50 vmov.f32 q1, #1 ; 0x3f800000
20 48: f2c0c050 vmov.i32 q14, #0 ; 0x00000000
21 4c: e1a00001 mov r0, r1
22 50: f464038f vld2.32 {d16-d19}, [r4]
23 54: e281b001 add fp, r1, #1
24 58: e281a002 add sl, r1, #2
25 5c: e2819003 add r9, r1, #3
26 60: f22081f0 vorr q4, q8, q8
27 64: e2844020 add r4, r4, #32
28 68: f22201f2 vorr q0, q9, q9
29 6c: f46e2a8f vld1.32 {d18-d19}, [lr]
30 70: f3e009c7 vmul.f32 q8, q8, d7[0]
31 74: f46c838f vld2.32 {d24-d27}, [ip]
32 78: f3e82146 vmla.f32 q9, q4, d6[0]
33 7c: f44eca8f vst1.32 {d28-d29}, [lr]
34 80: e28ee010 add lr, lr, #16
35 84: f3e00167 vmla.f32 q8, q0, d7[1]
36 88: f3e821e6 vmla.f32 q9, q12, d6[1]
37 8c: f3400eee vcge.f32 q8, q8, q15
38 90: f44d0a9f vst1.32 {d16-d17}, [sp :64]
39 94: e59d7000 ldr r7, [sp]
40 98: f26241f2 vorr q10, q9, q9
41 9c: f26021f0 vorr q9, q8, q8
42 a0: e59d6004 ldr r6, [sp, #4]
43 a4: e3570000 cmp r7, #0
44 a8: f352217c vbsl q9, q1, q14
45 ac: e59d5008 ldr r5, [sp, #8]
46 b0: 17821100 strne r1, [r2, r0, lsl #2]
47 b4: 12800001 addne r0, r0, #1
48 b8: e3560000 cmp r6, #0
49 bc: e2811004 add r1, r1, #4
50 c0: f3e02145 vmla.f32 q9, q0, d5[0]
51 c4: 1782b100 strne fp, [r2, r0, lsl #2]
52 c8: 12800001 addne r0, r0, #1
53 cc: e3550000 cmp r5, #0
54 d0: e59d500c ldr r5, [sp, #12]
55 d4: 1782a100 strne sl, [r2, r0, lsl #2]
56 d8: 12800001 addne r0, r0, #1
57 dc: e3550000 cmp r5, #0
58 e0: 17829100 strne r9, [r2, r0, lsl #2]
59 e4: 12800001 addne r0, r0, #1
60 e8: e1530001 cmp r3, r1
61 ec: f3ea21e5 vmla.f32 q9, q13, d5[1]
62 f0: f26261f2 vorr q11, q9, q9
63 f4: f44c438f vst2.32 {d20-d23}, [ip]
64 f8: e28cc020 add ip, ip, #32
65 fc: 8affffd3 bhi 50 <update_state_f32+0x50>
66 100: e9981010 ldmb r8, {r4, ip}
67 104: e588c004 str ip, [r8, #4]
68 108: e5884008 str r4, [r8, #8]
69 10c: e28dd014 add sp, sp, #20
70 110: ecbbd8b04 vpop {d8-d9}
71 114: e8bbd8ff0 pop {r4, r5, r6, r7, r8, r9, sl, fp, pc}
72 118: e1a00003 mov r0, r3
73 11c: eafffff8 b 104 <update_state_f32+0x104>

```

Figure C.4: update method Assembly code for the F32 NEON implementation.

```

1 00000000 <update_state_f32>:
2 0: e92d4ff0 push {r4, r5, r6, r7, r8, r9, sl, fp, lr}
3 4: e281c018 add ip, r1, #24
4 8: e281e008 add lr, r1, #8
5 c: e2814010 add r4, r1, #16
6 10: e3530000 cmp r3, #0
7 14: e1a08000 mov r8, r0
8 18: ed2d8b04 vpush {d8-d9}
9 1c: e24dd014 sub sp, sp, #20
10 20: f42e678f vld1.32 {d6}, [lr]
11 24: e590e000 ldr lr, [r0]
12 28: f424578f vld1.32 {d5}, [r4]
13 2c: e5904004 ldr r4, [r0, #4]

```

```

14 30: f4ececac vld1.32 {d30[]-d31[]}, [ip]
15 34: e590c008 ldr ip, [r0, #8]
16 38: f421778f vld1.32 {d7}, [r1]
17 3c: 0a00003b beq 130 <update_state_f32+0x130>
18 40: e3a01000 mov r1, #0
19 44: f2872f50 vmov.f32 q1, #1 ; 0x3f800000
20 48: f2c0c050 vmov.i32 q14, #0 ; 0x00000000
21 4c: e1a00001 mov r0, r1
22 50: e2845080 add r5, r4, #128 ; 0x80
23 54: f5d5f000 pld [r5]
24 58: e28c5080 add r5, ip, #128 ; 0x80
25 5c: f5d5f000 pld [r5]
26 60: e28e5020 add r5, lr, #32
27 64: f5d5f000 pld [r5]
28 68: f46403bf vld2.32 {d16-d19}, [r4 :256]
29 6c: e281b001 add fp, r1, #1
30 70: e281a002 add sl, r1, #2
31 74: e2819003 add r9, r1, #3
32 78: f22081f0 vorr q4, q8, q8
33 7c: e2844040 add r4, r4, #64 ; 0x40
34 80: f22201f2 vorr q0, q9, q9
35 84: f46e2aaf vld1.32 {d18-d19}, [lr :128]
36 88: f3e009c7 vmul.f32 q8, q8, d7[0]
37 8c: f46c83bf vld2.32 {d24-d27}, [ip :256]
38 90: f3e82146 vmla.f32 q9, q4, d6[0]
39 94: f44ecaaf vst1.32 {d28-d29}, [lr :128]
40 98: e28ee010 add lr, lr, #16
41 9c: f3e00167 vmla.f32 q8, q0, d7[1]
42 a0: f3e821e6 vmla.f32 q9, q12, d6[1]
43 a4: f3400eee vcge.f32 q8, q8, q15
44 a8: f44d0a9f vst1.32 {d16-d17}, [sp :64]
45 ac: e59d7000 ldr r7, [sp]
46 b0: f26241f2 vorr q10, q9, q9
47 b4: f26021f0 vorr q9, q8, q8
48 b8: e59d6004 ldr r6, [sp, #4]
49 bc: e3570000 cmp r7, #0
50 c0: f352217c vbsl q9, q1, q14
51 c4: e59d5008 ldr r5, [sp, #8]
52 c8: 17821100 strne r1, [r2, r0, lsl #2]
53 cc: 12800001 addne r0, r0, #1
54 d0: e3560000 cmp r6, #0
55 d4: e2811004 add r1, r1, #4
56 d8: f3e02145 vmla.f32 q9, q0, d5[0]
57 dc: 1782b100 strne fp, [r2, r0, lsl #2]
58 e0: 12800001 addne r0, r0, #1
59 e4: e3550000 cmp r5, #0
60 e8: e59d500c ldr r5, [sp, #12]
61 ec: 1782a100 strne sl, [r2, r0, lsl #2]
62 f0: 12800001 addne r0, r0, #1
63 f4: e3550000 cmp r5, #0
64 f8: 17829100 strne r9, [r2, r0, lsl #2]
65 fc: 12800001 addne r0, r0, #1
66 100: e1530001 cmp r3, r1
67 104: f3ea21e5 vmla.f32 q9, q13, d5[1]
68 108: f26261f2 vorr q11, q9, q9
69 10c: f44c43bf vst2.32 {d20-d23}, [ip :256]
70 110: e28cc040 add ip, ip, #64 ; 0x40
71 114: 8affffcd bhi 50 <update_state_f32+0x50>
72 118: e9981010 ldmbi r8, {r4, ip}
73 11c: e588c004 str ip, [r8, #4]
74 120: e5884008 str r4, [r8, #8]
75 124: e28dd014 add sp, sp, #20
76 128: ec8d8b04 vpop {d8-d9}
77 12c: e8bd8ff0 pop {r4, r5, r6, r7, r8, r9, sl, fp, pc}
78 130: e1a00003 mov r0, r3
79 134: eaafffff b 11c <update_state_f32+0x11c>

```

Figure C.5: update method Assembly code for the F32 memory optimized NEON implementation.

C.1.2 16-bit floating-point implementation

```

1 #include <arm_neon.h>
2
3 #define PLD(x)    asm ("pld [%0]" :: "r"(x))
4
5 #define ASSUME_ALIGNED(type, ptr) reinterpret_cast<type>(__builtin_assume_aligned(↵
    reinterpret_cast<void*>(ptr), 32))
6
7
8 unsigned int update_state(float** __restrict state, const float* __restrict filter, ↵
    unsigned int* output, const unsigned int len) // Restrict says pointers are not ↵
    overlapping
9 {
10     // Common registers between iterations
11
12     // Number of output spikes
13     unsigned int len_output = 0u;
14     // Load filter coefs + threshold
15     float16x8_t vfilt_f16 = vld1q_f16(filter);
16     float32x4_t vfilt_f32_0 = vcvt_f32_f16(vget_low_f16(vfilt_f16));
17     float32x4_t vfilt_f32_1 = vcvt_f32_f16(vget_high_f16(vfilt_f16));
18
19     // 63      32 31      0 : D register
20     // [forw1_nu forw1_eps]
21     float32x2_t vforw_duo = vget_low_f32(vfilt_f32_0);
22
23     // 63      32 31      0 : D register
24     // [back2_eps back1_eps]
25     float32x2_t vback_eps = vget_high_f32(vfilt_f32_0);
26
27     // 63      32 31      0 : D register
28     // [back2_nu  back1_nu ]
29     float32x2_t vback_nu  = vget_low_f32(vfilt_f32_1);
30
31     // 127 96 95 64 63 32 31  0 : Q register
32     // [theta theta theta theta]
33     float32x2_t vtheta_temp = vget_high_f32(vfilt_f32_1);
34     float32x4_t vtheta = vdupq_lane_f32(vtheta_temp, 0);
35
36     // 63      32 31      0 : D register
37     // [1.0f  1.0f  1.0f  1.0f ]
38     float32x4_t vones = vmovq_n_f32(1.0f);
39
40     // 63      32 31      0 : D register
41     // [0.0f  0.0f  0.0f  0.0f ]
42     float32x4_t vzeros = vmovq_n_f32(0.0f);
43     float16x4_t vzeros_f16 = vcvt_f16_f32(vzeros);
44
45     // Buffer array for the bit-mask allocated on the stack
46     uint32_t temp_output[4];
47
48     // Alias states
49     // input is loaded in a cache line but uses only 4 entries in the current iteration
50     __fp16* si{ASSUME_ALIGNED(__fp16*, state[0])};
51
52     // eps and nu are interleaved to use a cache line (8 words = 4 eps + 4 nu) (Optimal ↵
    for 4 // SIMD)
53     __fp16* s1{ASSUME_ALIGNED(__fp16*, state[1])};
54     __fp16* s2{ASSUME_ALIGNED(__fp16*, state[2])};
55
56     unsigned int n;
57     // Beginning of the cache line
58     __fp16* ss{(reinterpret_cast<unsigned int>(s1) < reinterpret_cast<unsigned int>(s2)?↵
    s1:s2)};
59     for(n = 0;
60         n < len;
61         n += 4, si += 4, s1 += 16, s2 += 16) // Number of states must be a multiple of 4
62     {
63         PLD(ss + 16);
64         PLD(si + 8);
65

```

```

66     float16x4x2_t vs1_f16 = vld2_f16(s1); // Deinterleave (eps | nu | eps | nu ...) ←
67         Uses half of the cache line
68     float16x4x2_t vs2_f16 = vld2_f16(s2);
69     float16x4_t vsi_f16 = vld1_f16(si);
70     vst1_f16(si, vzeros_f16); // Clear input (while still cached)
71
72     // Conversion
73     float32x4x2_t vs1;
74     float32x4x2_t vs2;
75     float32x4_t vsi;
76     vs1.val[0] = vcvt_f32_f16(vs1_f16.val[0]);
77     vs1.val[1] = vcvt_f32_f16(vs1_f16.val[1]);
78     vs2.val[0] = vcvt_f32_f16(vs2_f16.val[0]);
79     vs2.val[1] = vcvt_f32_f16(vs2_f16.val[1]);
80     vsi = vcvt_f32_f16(vsi_f16);
81
82     // Forward filter equation : u = forw1_eps * s1_eps + forw1_nu * s1_nu
83     // spike = (u >= theta)
84
85     // temp = f_1,eps * s_1,eps
86     float32x4_t vspike = vmulq_lane_f32(vs1.val[0], vforw_duo, 0);
87
88     // u = temp + f_1,nu * s_1,nu
89     vspike = vmlaq_lane_f32(vspike, vs1.val[1], vforw_duo, 1);
90
91     // mask = (u >= theta) (all 1 if true, all 0 otherwise)
92     uint32x4_t vcond = vcgeq_f32(vspike, vtheta);
93
94     // spike = 1.0 if mask all 1, 0.0 otherwise
95     vspike = vbslq_f32(vcond, vones, vzeros);
96
97     // Output spikes store
98     vst1q_u32(temp_output, vcond);
99     for (unsigned int m = 0; m < 4; ++m)
100         if (temp_output[m])
101             output[len_output++] = (n + m);
102
103     // Backward filter equation : s0_eps = input + back1_eps * s1_eps + back2_eps * s2_eps
104     // s0_nu = spike + back1_nu * s1_nu + back2_nu * s2_nu
105     float32x4x2_t vs0;
106
107     // Eps filter equation : temp_eps = input + back1_eps * s1_eps
108     vs0.val[0] = vmlaq_lane_f32(vsi, vs1.val[0], vback_eps, 0);
109
110     // Nu filter equation : temp_nu = spike + back1_nu * s1_nu
111     vs0.val[1] = vmlaq_lane_f32(vspike, vs1.val[1], vback_nu, 0);
112
113     // Eps filter equation : s0_eps = temp_eps + back2_eps * s2_eps
114     vs0.val[0] = vmlaq_lane_f32(vs0.val[0], vs2.val[0], vback_eps, 1);
115
116     // Nu filter equation : s0_nu = temp_nu + back2_nu * s2_nu
117     vs0.val[1] = vmlaq_lane_f32(vs0.val[1], vs2.val[1], vback_nu, 1);
118
119     // Conversion
120     float16x4x2_t vs0_f16;
121     vs0_f16.val[0] = vcvt_f16_f32(vs0.val[0]);
122     vs0_f16.val[1] = vcvt_f16_f32(vs0.val[1]);
123
124     // Store result in the s2 state memory
125     vst2_f16(s2, vs0_f16); // Interleave (eps | nu | eps | nu ...)
126 }
127
128 // Update states s2 ← s1 (change pointers head)
129 float32_t* temp = state[1];
130 state[1] = state[2];
131 state[2] = temp;
132
133 return len_output;

```

Figure C.6: update method for the F16 memory optimized NEON implementation.

```

1 00000000 <update_state_f16>:
2   0:   e590c000    ldr ip, [r0]
3   4:   e3530000    cmp r3, #0
4   8:   e92d4ff0    push {r4, r5, r6, r7, r8, r9, sl, fp, lr}
5   c:   e1a08000    mov r8, r0
6  10:   ed2d8b04    vpush {d8-d9}
7  14:   e24dd01c    sub sp, sp, #28
8  18:   f4610a4f    vld1.16 {d16-d17}, [r1]
9  1c:   e08c5083    add r5, ip, r3, lsl #1
10 20:   f3f62720    vcvf.f32.f16 q9, d16
11 24:   e5904004    ldr r4, [r0, #4]
12 28:   f3f60721    vcvf.f32.f16 q8, d17
13 2c:   e590e008    ldr lr, [r0, #8]
14 30:   f3f4ac61    vdup.32 q13, d17[0]
15 34:   0a000044    beq 14c <update_state_f16+0x14c>
16 38:   e3a01000    mov r1, #0
17 3c:   f22271b2    vorr d7, d18, d18
18 40:   f22361b3    vorr d6, d19, d19
19 44:   e1a00001    mov r0, r1
20 48:   f22051b0    vorr d5, d16, d16
21 4c:   f2c7cf50    vmov.f32 q14, #1 ; 0x3f800000
22 50:   f2c08050    vmov.i32 q12, #0 ; 0x00000000
23 54:   f3f64628    vcvf.f16.f32 d20, q12
24 58:   e58d8004    str r8, [sp, #4]
25 5c:   e2855020    add r5, r5, #32
26 60:   f5d5f000    pld [r5]
27 64:   e28c6010    add r6, ip, #16
28 68:   f5d6f000    pld [r6]
29 6c:   f464086f    vld2.16 {d16-d17}, [r4 :128]
30 70:   f3b60720    vcvf.f32.f16 q0, d16
31 74:   f3e02947    vmul.f32 q9, q0, d7[0]
32 78:   f3b62721    vcvf.f32.f16 q1, d17
33 7c:   f46e086f    vld2.16 {d16-d17}, [lr :128]
34 80:   f3b68720    vcvf.f32.f16 q4, d16
35 84:   f3f6e721    vcvf.f32.f16 q15, d17
36 88:   e28d6008    add r6, sp, #8
37 8c:   f46c075f    vld1.16 {d16}, [ip :64]
38 90:   f3f60720    vcvf.f32.f16 q8, d16
39 94:   f3e00146    vmla.f32 q8, q0, d6[0]
40 98:   e281b001    add fp, r1, #1
41 9c:   f44c475f    vst1.16 {d20}, [ip :64]
42 a0:   e281a002    add sl, r1, #2
43 a4:   f3e22167    vmla.f32 q9, q1, d7[1]
44 a8:   e2819003    add r9, r1, #3
45 ac:   e2844020    add r4, r4, #32
46 b0:   e28cc008    add ip, ip, #8
47 b4:   f3e80166    vmla.f32 q8, q4, d6[1]
48 b8:   f3f66620    vcvf.f16.f32 d22, q8
49 bc:   f3420eea    vcge.f32 q8, q9, q13
50 c0:   f26021f0    vorr q9, q8, q8
51 c4:   f4460a9f    vst1.32 {d16-d17}, [r6 :64]
52 c8:   f35c21f8    vbsl q9, q14, q12
53 cc:   e59d8008    ldr r8, [sp, #8]
54 d0:   e59d700c    ldr r7, [sp, #12]
55 d4:   e59d6010    ldr r6, [sp, #16]
56 d8:   f26201f2    vorr q8, q9, q9
57 dc:   e3580000    cmp r8, #0
58 e0:   17821100    strne r1, [r2, r0, lsl #2]
59 e4:   12800001    addne r0, r0, #1
60 e8:   e3570000    cmp r7, #0
61 ec:   e2811004    add r1, r1, #4
62 f0:   f3e20145    vmla.f32 q8, q1, d5[0]
63 f4:   1782b100    strne fp, [r2, r0, lsl #2]
64 f8:   12800001    addne r0, r0, #1
65 fc:   e3560000    cmp r6, #0
66 100: e59d6014    ldr r6, [sp, #20]
67 104: 1782a100    strne sl, [r2, r0, lsl #2]
68 108: 12800001    addne r0, r0, #1
69 10c: e3560000    cmp r6, #0
70 110: 17829100    strne r9, [r2, r0, lsl #2]
71 114: 12800001    addne r0, r0, #1
72 118: e1530001    cmp r3, r1
73 11c: f3ee01e5    vmla.f32 q8, q15, d5[1]
74 120: f3f67620    vcvf.f16.f32 d23, q8

```

```

75 124: f44e686f vst2.16 {d22-d23}, [lr :128]
76 128: e28ee020 add lr, lr, #32
77 12c: 8affffc8 bhi 5c <update_state_f16+0x5c>
78 130: e59d8004 ldr r8, [sp, #4]
79 134: e9984010 ldmbi r8, {r4, lr}
80 138: e588e004 str lr, [r8, #4]
81 13c: e5884008 str r4, [r8, #8]
82 140: e28dd01c add sp, sp, #28
83 144: ec8db004 vpop {d8-d9}
84 148: e8bd8ff0 pop {r4, r5, r6, r7, r8, r9, sl, fp, pc}
85 14c: e1a00003 mov r0, r3
86 150: eafffff8 b 138 <update_state_f16+0x138>

```

Figure C.7: update method Assembly code for the F16 memory optimized NEON implementation.

C.2 Dense operation method

C.2.1 32-bit floating-point implementation

```

1 #define PLD(x) asm ("pld [%0]"::"r"(x))
2 #define ASSUME_ALIGNED(type, ptr) reinterpret_cast<type>(__builtin_assume_aligned(↵
   reinterpret_cast<void*>(ptr), 32))
3
4 template <unsigned int outFeatures>
5 void dense_operation(float* __restrict input, float* __restrict weight, Spike_s *spike)
6 {
7     unsigned int event;
8     float *rrinput{ASSUME_ALIGNED(float*, input)};
9     float *rrweight{ASSUME_ALIGNED(float*, weight)};
10
11     for (event = 0; event < spike->len_spike; ++event)
12     {
13         unsigned int n;
14         float *rrweight;
15         float *rrinput;
16
17         for (n = 0, rrweight = rweight + (outFeatures * spike->array_spike[event]), ↵
           rrinput = rinput;
18             n < outFeatures;
19             n += 8, rrweight += 8, rrinput += 8)
20         {
21             PLD(rrinput + 8);
22             PLD(rrweight + 8);
23             float32x4x2_t vinput = vld2q_f32(rrinput);
24             float32x4x2_t vweight = vld2q_f32(rrweight);
25             vinput.val[0] = vaddq_f32(vinput.val[0], vweight.val[0]);
26             vinput.val[1] = vaddq_f32(vinput.val[1], vweight.val[1]);
27             vst2q_f32(rrinput, vinput);
28         }
29     }
30 }

```

Figure C.8: Sparse dense_operation method with F32 NEON intrinsics.

```

1 00000000 <dense_operation_f32>:
2 0: e5923004 ldr r3, [r2, #4]
3 4: e3530000 cmp r3, #0
4 8: 012fff1e bxeq lr
5 c: e92d4070 push {r4, r5, r6, lr}

```

```

6   10:  e3a06000    mov r6, #0
7   14:  e5923000    ldr r3, [r2]
8   18:  e1a0c000    mov ip, r0
9   1c:  e7933106    ldr r3, [r3, r6, lsl #2]
10  20:  e0813483    add r3, r1, r3, lsl #9
11  24:  e2835c02    add r5, r3, #512      ; 0x200
12  28:  e28ce020    add lr, ip, #32
13  2c:  f5def000    pld [lr]
14  30:  e2834020    add r4, r3, #32
15  34:  f5d4f000    pld [r4]
16  38:  f46c83bf    vld2.32 {d24-d27}, [ip :256]
17  3c:  e1540005    cmp r4, r5
18  40:  f46343bf    vld2.32 {d20-d23}, [r3 :256]
19  44:  e1a03004    mov r3, r4
20  48:  f2480de4    vadd.f32    q8, q12, q10
21  4c:  f24a2de6    vadd.f32    q9, q13, q11
22  50:  f44c03bf    vst2.32 {d16-d19}, [ip :256]
23  54:  e1a0c00e    mov ip, lr
24  58:  1afffff2    bne 28 <dense_operation_f32+0x28>
25  5c:  e5923004    ldr r3, [r2, #4]
26  60:  e2866001    add r6, r6, #1
27  64:  e1530006    cmp r3, r6
28  68:  8affffe9    bhi 14 <dense_operation_f32+0x14>
29  6c:  e8bd8070    pop {r4, r5, r6, pc}

```

Figure C.9: Sparse dense_operation method Assembly code for the F32 NEON intrinsic implementation.

C.2.2 16-bit floating-point implementation

```

1  #define PLD(x)  asm ("pld [%0]"::"r"(x))
2  #define PLDW(x) asm ("pldw [%0]"::"r"(x))
3  #define ASSUME_ALIGNED(type, ptr) reinterpret_cast<type>(__builtin_assume_aligned(↵
    reinterpret_cast<void*>(ptr), 32))
4
5  template <unsigned int outFeatures>
6  void dense_operation(__fp16* __restrict input, __fp16* __restrict weight, Spike_s *spike)
7  {
8      unsigned int tile;
9      unsigned int event;
10     __fp16 *rinput{ASSUME_ALIGNED(__fp16*, input)};
11     __fp16 *rweight{ASSUME_ALIGNED(__fp16*, weight)};
12
13     for (tile = 0;
14          tile < outFeatures;
15          tile += 16, rinput += 16, rweight += 16)    // Load a cache line (32 bytes = 16 ↵
        __fp16)
16
17         // Load fp16
18         float16x8x2_t vinput_f16 = vld2q_f16(rinput);
19
20         // Conversion
21         float32x4_t vinput__0 = vcvt_f32_f16(vget_low_f16( vinput_f16.val[0]));
22         float32x4_t vinput__1 = vcvt_f32_f16(vget_high_f16(vinput_f16.val[0]));
23         float32x4_t vinput__2 = vcvt_f32_f16(vget_low_f16( vinput_f16.val[1]));
24         float32x4_t vinput__3 = vcvt_f32_f16(vget_high_f16(vinput_f16.val[1]));
25
26         for (event = 0; event < spike->len_spike; ++event)
27         {
28             PLD(rweight + (outFeatures * spike->array_spike[event + 2]));
29             PLDW(rinput);
30             PLD(rinput + 16);
31             // Load fp16
32             __fp16 *rrweight{rweight + (outFeatures * spike->array_spike[event])};
33             float16x8x2_t vweight_f16 = vld2q_f16(rrweight);
34
35             // Conversion
36             float32x4_t vweight__0 = vcvt_f32_f16(vget_low_f16( vweight_f16.val[0]));

```

```

37     float32x4_t vweight__1 = vcvf32_f16(vget_high_f16(vweight_f16.val[0]));
38     float32x4_t vweight__2 = vcvf32_f16(vget_low_f16(vweight_f16.val[1]));
39     float32x4_t vweight__3 = vcvf32_f16(vget_high_f16(vweight_f16.val[1]));
40
41     vinput__0 = vaddq_f32(vinput__0, vweight__0);
42     vinput__1 = vaddq_f32(vinput__1, vweight__1);
43     vinput__2 = vaddq_f32(vinput__2, vweight__2);
44     vinput__3 = vaddq_f32(vinput__3, vweight__3);
45 }
46
47 // Conversion
48 float16x4_t vtemp__0, vtemp__1;
49 vtemp__0 = vcvf16_f32(vinput__0);
50 vtemp__1 = vcvf16_f32(vinput__1);
51 vinput_f16.val[0] = vcombine_f16(vtemp__0, vtemp__1);
52 vtemp__0 = vcvf16_f32(vinput__2);
53 vtemp__1 = vcvf16_f32(vinput__3);
54 vinput_f16.val[1] = vcombine_f16(vtemp__0, vtemp__1);
55
56 vst2q_f16(rinput, vinput_f16);
57 }
58 }

```

Figure C.10: Sparse dense_operation method with F16 NEON intrinsics.

```

1 00000000 <dense_operation_f16>:
2   0: e92d40f0 push {r4, r5, r6, r7, lr}
3   4: e2800020 add r0, r0, #32
4   8: e2817c01 add r7, r1, #256 ; 0x100
5   c: e2405020 sub r5, r0, #32
6  10: e5926004 ldr r6, [r2, #4]
7  14: f465c34f vld2.16 {d28-d31}, [r5]
8  18: f3f6a72c vcvf32.f16 q13, d28
9  1c: f3f6872d vcvf32.f16 q12, d29
10 20: f3f6672e vcvf32.f16 q11, d30
11 24: e3560000 cmp r6, #0
12 28: f3f6472f vcvf32.f16 q10, d31
13 2c: 0a000016 beq 8c <dense_operation_f16+0x8c>
14 30: e5924000 ldr r4, [r2]
15 34: e3a0e008 mov lr, #8
16 38: e3a03000 mov r3, #0
17 3c: e794c00e ldr ip, [r4, lr]
18 40: e081c40c add ip, r1, ip, lsl #8
19 44: f5dcf000 pld [ip]
20 48: f595f000 pldw [r5]
21 4c: f5d0f000 pld [r0]
22 50: e794c103 ldr ip, [r4, r3, lsl #2]
23 54: e2833001 add r3, r3, #1
24 58: e1560003 cmp r6, r3
25 5c: e28ee004 add lr, lr, #4
26 60: e081c40c add ip, r1, ip, lsl #8
27 64: f46c037f vld2.16 {d16-d19}, [ip :256]
28 68: f3b66721 vcvf32.f16 q3, d17
29 6c: f3b64720 vcvf32.f16 q2, d16
30 70: f2488dc6 vadd.f32 q12, q12, q3
31 74: f3f60723 vcvf32.f16 q8, d19
32 78: f3b66722 vcvf32.f16 q3, d18
33 7c: f24aad4c vadd.f32 q13, q13, q2
34 80: f2466dc6 vadd.f32 q11, q11, q3
35 84: f2444de0 vadd.f32 q10, q10, q8
36 88: 8affffeb bhi 3c <dense_operation_f16+0x3c>
37 8c: f3f6a62a vcvf16.f32 d26, q13
38 90: f3f68628 vcvf16.f32 d24, q12
39 94: e2811020 add r1, r1, #32
40 98: f26ac1ba vorr d28, d26, d26
41 9c: f268d1b8 vorr d29, d24, d24
42 a0: f3f66626 vcvf16.f32 d22, q11
43 a4: f3f64624 vcvf16.f32 d20, q10
44 a8: f266e1b6 vorr d30, d22, d22
45 ac: f264f1b4 vorr d31, d20, d20
46 b0: e1570001 cmp r7, r1

```

```

47 b4: e2800020    add r0, r0, #32
48 b8: f445c34f    vst2.16 {d28-d31}, [r5]
49 bc: 1affffd2    bne c <dense_operation_f16+0xc>
50 c0: e8bd80f0    pop {r4, r5, r6, r7, pc}

```

Figure C.11: Sparse dense_operation method Assembly code for the F16 NEON intrinsic implementation.

C.3 Convolution operation method

```

1  template <typename Type, unsigned int inWidth, unsigned int inHeight, unsigned int ←
    inChannels, unsigned outChannels>
2  void conv_operation(Type* __restrict input, Type* __restrict weight, Spike_s *spike)
3  {
4      constexpr unsigned int outWidth{LEN_CONV(inWidth)};
5      constexpr unsigned int h2{(inHeight + 1) / 2};
6      constexpr unsigned int w2{(inWidth + 1) / 2};
7
8      // Convolution operation (Weights are separated into even / odd parts)
9      unsigned int event;
10     for (event = 0; event < spike->len_spike; ++event)
11     {
12         // Compute spatial indexes from global index
13         // i_e <=> [h_e, w_e, c_e]
14         const unsigned int i_e {spike->array_spike[event]};
15         // Compile time divisions => shift + mul instr
16         const unsigned int temp {i_e / inChannels};
17         const unsigned int c_e {i_e - temp * inChannels};
18         const unsigned int h_e {temp / inWidth};
19         const unsigned int w_e {temp - h_e * inWidth};
20
21         //
22         // If the input event has an even/odd spatial index, it only uses the even/odd ←
23         // part of
24         // the filter (stride = 2 by default).
25         // Even - Odd test + Border test
26         const unsigned int h_e2 {(h_e + 1) / 2};
27         const bool is_h_even {h_e % 2 == 0};
28         const unsigned int kh2 {(is_h_even)?LEN_KERNEL_2_B:LEN_KERNEL_2_S};
29
30         const int h_e_offset_left {((int) h_e2) - ((int) LEN_KERNEL_2_S)};
31         const bool h_not_border_left {h_e_offset_left > 0};
32         const unsigned int h_y_start {h_not_border_left?h_e_offset_left:0u};
33         const unsigned int h_w_start {h_y_start - h_e_offset_left};
34         const unsigned int h_e_offset_right {h2 - h_e2};
35         const unsigned int h_len {(h_e_offset_right > kh2)?kh2:←
36             h_e_offset_right) - h_w_start};
37
38         const unsigned int w_e2 {(w_e + 1) / 2};
39         const bool is_w_even {w_e % 2 == 0};
40         const unsigned int kw2 {(is_w_even)?LEN_KERNEL_2_B:LEN_KERNEL_2_S};
41
42         const int w_e_offset_left {((int) w_e2) - ((int) LEN_KERNEL_2_S)};
43         const bool w_not_border_left {w_e_offset_left > 0};
44         const unsigned int w_y_start {w_not_border_left?w_e_offset_left:0u};
45         const unsigned int w_w_start {w_y_start - w_e_offset_left};
46         const unsigned int w_e_offset_right {w2 - w_e2};
47         const unsigned int w_len {(w_e_offset_right > kw2)?kw2:←
48             w_e_offset_right) - w_w_start};
49
50         unsigned int h, w;
51         Type *rinput, *rweight, *rrinput, *rrweight;
52         for (h = 0,
            rinput = &input[outChannels * (outWidth * h_y_start + w_y_start)],
            // Inlined access function of the even/odd part of the filter

```

```

53         rweight = &weight[weight_index(c_e, h_w_start, w_w_start, is_h_even, ←
54             is_w_even, outChannels)];
55         h < h_len;
56         ++h,
57         rinput += (outChannels * outWidth),
58         rweight += (outChannels * kw2))
59     {
60         for (w = 0, rrinput = rinput, rrweight = rweight;
61             w < w_len;
62             ++w,
63             rrinput += outChannels,
64             rrweight += outChannels)
65         {
66             // Inlined Accumulate function
67             vadd(rrinput, rrweight, outChannels);
68         }
69     }
70 }

```

Figure C.12: Sparse conv_operation method template for any data-type.

```

1  template <typename Type, unsigned int inWidth, unsigned int inHeight, unsigned int ←
2      inChannels, unsigned outChannels>
3  void conv_operation(Type* __restrict input, Type* __restrict weight, Spike_s *spike)
4  {
5      constexpr unsigned int outWidth{LEN_CONV(inWidth)};
6      constexpr unsigned int h2{(inHeight + 1) / 2};
7      constexpr unsigned int w2{(inWidth + 1) / 2};
8
9      // Convolution operation (Weights are separated into even / odd parts)
10     unsigned int event;
11     for (event = 0; event < spike->len_spike; ++event)
12     {
13         // Compute spatial indexes from global index
14         // i_e <=> [h_e, w_e, c_e]
15         const unsigned int i_e {spike->array_spike[event]};
16         // Compile time divisions => shift + mul instr
17         const unsigned int temp {i_e / inChannels};
18         const unsigned int c_e {i_e - temp * inChannels};
19         const unsigned int h_e {temp / inWidth};
20         const unsigned int w_e {temp - h_e * inWidth};
21
22         //
23         // If the input event has an even/odd spatial index, it only uses the even/odd ←
24         // part of
25         // the filter (stride = 2 by default).
26         //
27         // Even - Odd test + Border test
28         const unsigned int h_e2 {(h_e + 1) / 2};
29         const bool is_h_even {h_e % 2 == 0};
30         const unsigned int kh2 {(is_h_even)?LEN_KERNEL_2_B:LEN_KERNEL_2_S};
31
32         const int h_e_offset_left {((int) h_e2) - ((int) LEN_KERNEL_2_S)};
33         const bool h_not_border_left {h_e_offset_left > 0};
34         const unsigned int h_y_start {h_not_border_left?h_e_offset_left:0u};
35         const unsigned int h_w_start {h_y_start - h_e_offset_left};
36         const unsigned int h_e_offset_right {h2 - h_e2};
37         const unsigned int h_len {((h_e_offset_right > kh2)?kh2:←
38             h_e_offset_right) - h_w_start};
39
40         const unsigned int w_e2 {(w_e + 1) / 2};
41         const bool is_w_even {w_e % 2 == 0};
42         const unsigned int kw2 {(is_w_even)?LEN_KERNEL_2_B:LEN_KERNEL_2_S};
43
44         const int w_e_offset_left {((int) w_e2) - ((int) LEN_KERNEL_2_S)};
45         const bool w_not_border_left {w_e_offset_left > 0};
46         const unsigned int w_y_start {w_not_border_left?w_e_offset_left:0u};
47         const unsigned int w_w_start {w_y_start - w_e_offset_left};
48         const unsigned int w_e_offset_right {w2 - w_e2};

```

```

46     const unsigned int w_len      {((w_e_offset_right > kw2)?kw2:(w_e_offset_right) - w_w_start)};
47
48     unsigned int h, w;
49     Type *rinput, *rweight, *rrinput, *rrweight;
50     for (h = 0,
51          rinput = &input[outChannels * (outWidth * h_y_start + w_y_start)],
52          // Inlined access function of the even/odd part of the filter
53          rweight = &weight[weight_index(c_e, h_w_start, w_w_start, is_h_even, is_w_even, outChannels)];
54
55     h < h_len;
56     ++h,
57     rinput += (outChannels * outWidth),
58     rweight += (outChannels * kw2))
59     {
60         for (w = 0, rrinput = rinput, rrweight = rweight;
61              w < w_len;
62              ++w,
63              rrinput += outChannels,
64              rrweight += outChannels)
65         {
66             // Inlined Accumulate function
67             vadd(rrinput, rrweight, outChannels);
68         }
69     }
70 }

```

Figure C.13: Inlined even/Odd indexing method for the sparse convolution.

C.3.1 32-bit floating-point implementation

```

1  inline void vadd(float *__restrict input, float *__restrict weight, const unsigned int len)
2  {
3      unsigned int c;
4      // len is a multiple of 8 (use full cache line)
5      float *rinput{ASSUME_ALIGNED(float*, input)};
6      float *rweight{ASSUME_ALIGNED(float*, weight)};
7      for (c = 0;
8           c < len;
9           c += 8, rinput += 8, rweight += 8)
10     {
11         PLD(rinput + 8);
12         PLD(rweight + 8);
13
14         // Load interleaving
15         float32x4x2_t vinput = vld2q_f32(rinput);
16         float32x4x2_t vweight = vld2q_f32(rweight);
17
18         // Add
19         vinput.val[0] = vaddq_f32(vinput.val[0], vweight.val[0]);
20         vinput.val[1] = vaddq_f32(vinput.val[1], vweight.val[1]);
21
22         // Store interleaving
23         vst2q_f32(rinput, vinput);
24     }
25 }

```

Figure C.14: Inlined accumulate method for the sparse convolution with F32 NEON intrinsics.

```

1  00000000 <conv_operation_f32>:
2  0:  e92d4ff0  push    {r4, r5, r6, r7, r8, r9, sl, fp, lr}

```

```

3      4: e24dd014 sub sp, sp, #20
4      8: e5924004 ldr r4, [r2, #4]
5      c: e58d2004 str r2, [sp, #4]
6     10: e3540000 cmp r4, #0
7     14: e58d0008 str r0, [sp, #8]
8     18: e58d100c str r1, [sp, #12]
9     1c: 0a000061 beq 1a8 <conv_operation_f32+0x1a8>
10    20: e308b51f movw fp, #34079 ; 0x851f
11    24: e3a03000 mov r3, #0
12    28: e345b1eb movt fp, #20971 ; 0x51eb
13    2c: e58d3000 str r3, [sp]
14    30: e59d3004 ldr r3, [sp, #4]
15    34: e59d2000 ldr r2, [sp]
16    38: e5933000 ldr r3, [r3]
17    3c: e7932102 ldr r2, [r3, r2, lsl #2]
18    40: e3a03d19 mov r3, #1600 ; 0x640
19    44: e202603f and r6, r2, #63 ; 0x3f
20    48: e0810b92 umull r0, r1, r2, fp
21    4c: e0060693 mul r6, r3, r6
22    50: e1a034a1 lsr r3, r1, #9
23    54: e0831083 add r1, r3, r3, lsl #1
24    58: e2839001 add r9, r3, #1
25    5c: e1a090a9 lsr r9, r9, #1
26    60: e213e001 ands lr, r3, #1
27    64: e0831181 add r1, r3, r1, lsl #3
28    68: 13a08002 movne r8, #2
29    6c: e2493002 sub r3, r9, #2
30    70: e0611322 rsb r1, r1, r2, lsr #6
31    74: e1c30fc3 bic r0, r3, r3, asr #31
32    78: e2815001 add r5, r1, #1
33    7c: e0807080 add r7, r0, r0, lsl #1
34    80: e1a050a5 lsr r5, r5, #1
35    84: e0403003 sub r3, r0, r3
36    88: e0600107 rsb r0, r0, r7, lsl #2
37    8c: 03a08003 moveq r8, #3
38    90: e2452002 sub r2, r5, #2
39    94: e1c2cfc2 bic ip, r2, r2, asr #31
40    98: e2011001 and r1, r1, #1
41    9c: e269900a rsb r9, r9, #10
42    a0: e04c2002 sub r2, ip, r2
43    a4: e080000c add r0, r0, ip
44    a8: e1590008 cmp r9, r8
45    ac: 90639009 rsbls r9, r3, r9
46    b0: 80639008 rsbhi r9, r3, r8
47    b4: e59dc008 ldr ip, [sp, #8]
48    b8: e3510000 cmp r1, #0
49    bc: e265500d rsb r5, r5, #13
50    c0: 13a07002 movne r7, #2
51    c4: 03a07003 moveq r7, #3
52    c8: 13a0ac02 movne sl, #512 ; 0x200
53    cc: 03a0ac03 moveq sl, #768 ; 0x300
54    d0: e1550007 cmp r5, r7
55    d4: 90625005 rsbls r5, r2, r5
56    d8: 80625007 rsbhi r5, r2, r7
57    dc: e35e0000 cmp lr, #0
58    e0: e08c7400 add r7, ip, r0, lsl #8
59    e4: 1a000031 bne 1b0 <conv_operation_f32+0x1b0>
60    e8: e3510000 cmp r1, #0
61    ec: 10823083 addne r3, r2, r3, lsl #1
62    f0: 00833083 addeq r3, r3, r3, lsl #1
63    f4: 12833009 addne r3, r3, #9
64    f8: 00832002 addeq r2, r3, r2
65    fc: 00866302 addeq r6, r6, r2, lsl #6
66 100: 10866303 addne r6, r6, r3, lsl #6
67 104: e59d300c ldr r3, [sp, #12]
68 108: e3590000 cmp r9, #0
69 10c: e0836106 add r6, r3, r6, lsl #2
70 110: 0a00001f beq 194 <conv_operation_f32+0x194>
71 114: e3550000 cmp r5, #0
72 118: 0a00001d beq 194 <conv_operation_f32+0x194>
73 11c: e3a08000 mov r8, #0
74 120: e1a0e006 mov lr, r6
75 124: e1a03007 mov r3, r7
76 128: e3a0c000 mov ip, #0

```

```

77 12c: e1a0200e    mov r2, lr
78 130: e2834c01    add r4, r3, #256      ; 0x100
79 134: e2831020    add r1, r3, #32
80 138: f5d1f000    pld [r1]
81 13c: e2820020    add r0, r2, #32
82 140: f5d0f000    pld [r0]
83 144: f46383bf    vld2.32 {d24-d27}, [r3 :256]
84 148: e1510004    cmp r1, r4
85 14c: f46243bf    vld2.32 {d20-d23}, [r2 :256]
86 150: e1a02000    mov r2, r0
87 154: f2480de4    vadd.f32     q8, q12, q10
88 158: f24a2de6    vadd.f32     q9, q13, q11
89 15c: f44303bf    vst2.32 {d16-d19}, [r3 :256]
90 160: e1a03001    mov r3, r1
91 164: 1afffff2    bne 134 <conv_operation_f32+0x134>
92 168: e28cc001    add ip, ip, #1
93 16c: e28eec01    add lr, lr, #256      ; 0x100
94 170: e155000c    cmp r5, ip
95 174: 1affffec    bne 12c <conv_operation_f32+0x12c>
96 178: e2888001    add r8, r8, #1
97 17c: e2877c0b    add r7, r7, #2816     ; 0xb00
98 180: e1590008    cmp r9, r8
99 184: e086600a    add r6, r6, sl
100 188: 1affffe4    bne 120 <conv_operation_f32+0x120>
101 18c: e59d3004    ldr r3, [sp, #4]
102 190: e5934004    ldr r4, [r3, #4]
103 194: e59d3000    ldr r3, [sp]
104 198: e2833001    add r3, r3, #1
105 19c: e1540003    cmp r4, r3
106 1a0: e58d3000    str r3, [sp]
107 1a4: 8affffa1    bhi 30 <conv_operation_f32+0x30>
108 1a8: e28dd014    add sp, sp, #20
109 1ac: e8bd8ff0    pop {r4, r5, r6, r7, r8, r9, sl, fp, pc}
110 1b0: e3510000    cmp r1, #0
111 1b4: 00833083    addeq r3, r3, r3, lsl #1
112 1b8: 10823083    addne r3, r2, r3, lsl #1
113 1bc: 0283300f    addeq r3, r3, #15
114 1c0: 12833015    addne r3, r3, #21
115 1c4: 00832002    addeq r2, r3, r2
116 1c8: 10866303    addne r6, r6, r3, lsl #6
117 1cc: 00866302    addeq r6, r6, r2, lsl #6
118 1d0: eaffffcbb    b 104 <conv_operation_f32+0x104>

```

Figure C.15: Sparse conv_operation method Assembly code for the F32 NEON implementation.

C.3.2 16-bit floating-point implementation

```

1 inline void vadd(__fp16 *__restrict input, __fp16 *__restrict weight, const unsigned int len)
2 {
3     unsigned int c;
4     __fp16 *rinp{ASSUME_ALIGNED(__fp16*, input)};
5     __fp16 *rweight{ASSUME_ALIGNED(__fp16*, weight)};
6
7     // len is a multiple of 16 (use full cache line)
8     for (c = 0; c < len; c += 16, rinp += 16, rweight += 16)
9     {
10         PLD(rinp + 16);
11         PLD(rweight + 16);
12
13         // Load interleaving
14         float16x8x2_t vinput_f16 = vld2q_f16(rinp);
15         float16x8x2_t vweight_f16 = vld2q_f16(rweight);
16
17         // Convert
18         float32x4_t vinput__0 = vcvt_f32_f16(vget_low_f16(vinput_f16.val[0]));
19         float32x4_t vinput__1 = vcvt_f32_f16(vget_high_f16(vinput_f16.val[0]));

```

```

20 float32x4_t vinput__2 = vcvt_f32_f16(vget_low_f16( vinput_f16.val[1]));
21 float32x4_t vinput__3 = vcvt_f32_f16(vget_high_f16( vinput_f16.val[1]));
22 float32x4_t vweight__0 = vcvt_f32_f16(vget_low_f16( vweight_f16.val[0]));
23 float32x4_t vweight__1 = vcvt_f32_f16(vget_high_f16(vweight_f16.val[0]));
24 float32x4_t vweight__2 = vcvt_f32_f16(vget_low_f16( vweight_f16.val[1]));
25 float32x4_t vweight__3 = vcvt_f32_f16(vget_high_f16(vweight_f16.val[1]));
26
27 // Add
28 vinput__0 = vaddq_f32(vinput__0, vweight__0);
29 vinput__1 = vaddq_f32(vinput__1, vweight__1);
30 vinput__2 = vaddq_f32(vinput__2, vweight__2);
31 vinput__3 = vaddq_f32(vinput__3, vweight__3);
32
33 // Conversion
34 float16x4_t vtemp__0, vtemp__1;
35 vtemp__0 = vcvt_f16_f32(vinput__0);
36 vtemp__1 = vcvt_f16_f32(vinput__1);
37 vinput_f16.val[0] = vcombine_f16(vtemp__0, vtemp__1);
38 vtemp__0 = vcvt_f16_f32(vinput__2);
39 vtemp__1 = vcvt_f16_f32(vinput__3);
40 vinput_f16.val[1] = vcombine_f16(vtemp__0, vtemp__1);
41
42 // Store interleaving
43 vst2q_f16(rinput, vinput_f16);
44 }
45 }

```

Figure C.16: Inlined accumulate method for the sparse convolution with F16 NEON intrinsics.

```

1 00000000 <conv_operation_f16>:
2 0: e92d4ff0 push {r4, r5, r6, r7, r8, r9, sl, fp, lr}
3 4: e24dd014 sub sp, sp, #20
4 8: e5924004 ldr r4, [r2, #4]
5 c: e58d2004 str r2, [sp, #4]
6 10: e3540000 cmp r4, #0
7 14: e58d0008 str r0, [sp, #8]
8 18: e58d100c str r1, [sp, #12]
9 1c: 0a000073 beq lf0 <conv_operation_f16+0x1f0>
10 20: e308b51f movw fp, #34079 ; 0x851f
11 24: e3a03000 mov r3, #0
12 28: e345b1eb movt fp, #20971 ; 0x51eb
13 2c: e58d3000 str r3, [sp]
14 30: e59d3004 ldr r3, [sp, #4]
15 34: e59d2000 ldr r2, [sp]
16 38: e5933000 ldr r3, [r3]
17 3c: e7932102 ldr r2, [r3, r2, lsl #2]
18 40: e3a03d19 mov r3, #1600 ; 0x640
19 44: e202603f and r6, r2, #63 ; 0x3f
20 48: e0810b92 umull r0, r1, r2, fp
21 4c: e0060693 mul r6, r3, r6
22 50: e1a034a1 lsr r3, r1, #9
23 54: e0831083 add r1, r3, r3, lsl #1
24 58: e2839001 add r9, r3, #1
25 5c: e1a090a9 lsr r9, r9, #1
26 60: e213e001 ands lr, r3, #1
27 64: e0831181 add r1, r3, r1, lsl #3
28 68: 13a08002 movne r8, #2
29 6c: e2493002 sub r3, r9, #2
30 70: e0611322 rsb r1, r1, r2, lsr #6
31 74: e1c30fc3 bic r0, r3, r3, asr #31
32 78: e2815001 add r5, r1, #1
33 7c: e0807080 add r7, r0, r0, lsl #1
34 80: e1a050a5 lsr r5, r5, #1
35 84: e0403003 sub r3, r0, r3
36 88: e0600107 rsb r0, r0, r7, lsl #2
37 8c: 03a08003 moveq r8, #3
38 90: e2452002 sub r2, r5, #2
39 94: e1c2cfc2 bic ip, r2, r2, asr #31
40 98: e2011001 and r1, r1, #1

```

```

41 9c: e269900a rsb r9, r9, #10
42 a0: e04c2002 sub r2, ip, r2
43 a4: e080000c add r0, r0, ip
44 a8: e1590008 cmp r9, r8
45 ac: 90639009 rsbls r9, r3, r9
46 b0: 80639008 rsbhi r9, r3, r8
47 b4: e59dc008 ldr ip, [sp, #8]
48 b8: e3510000 cmp r1, #0
49 bc: e265500d rsb r5, r5, #13
50 c0: 13a07002 movne r7, #2
51 c4: 03a07003 moveq r7, #3
52 c8: 13a0ac01 movne sl, #256 ; 0x100
53 cc: 03a0ad06 moveq sl, #384 ; 0x180
54 d0: e1550007 cmp r5, r7
55 d4: 90625005 rsbls r5, r2, r5
56 d8: 80625007 rsbhi r5, r2, r7
57 dc: e35e0000 cmp lr, #0
58 e0: e08c7380 add r7, ip, r0, lsl #7
59 e4: 1a000043 bne lf8 <conv_operation_f16+0x1f8>
60 e8: e3510000 cmp r1, #0
61 ec: 10823083 addne r3, r2, r3, lsl #1
62 f0: 00833083 addeq r3, r3, r3, lsl #1
63 f4: 12833009 addne r3, r3, #9
64 f8: 00832002 addeq r2, r3, r2
65 fc: 00866302 addeq r6, r6, r2, lsl #6
66 100: 10866303 addne r6, r6, r3, lsl #6
67 104: e59d300c ldr r3, [sp, #12]
68 108: e3590000 cmp r9, #0
69 10c: e0836086 add r6, r3, r6, lsl #1
70 110: 0a000031 beq ldc <conv_operation_f16+0x1dc>
71 114: e3550000 cmp r5, #0
72 118: 0a00002f beq ldc <conv_operation_f16+0x1dc>
73 11c: e3a08000 mov r8, #0
74 120: e1a0e006 mov lr, r6
75 124: e1a03007 mov r3, r7
76 128: e3a0c000 mov ip, #0
77 12c: e1a0200e mov r2, lr
78 130: e2834080 add r4, r3, #128 ; 0x80
79 134: e2831020 add r1, r3, #32
80 138: f5d1f000 pld [r1]
81 13c: e2820020 add r0, r2, #32
82 140: f5d0f000 pld [r0]
83 144: f463037f vld2.16 {d16-d19}, [r3 :256]
84 148: f3f68721 vcvf.f32.f16 q12, d17
85 14c: f3f6e720 vcvf.f32.f16 q15, d16
86 150: f3f6a722 vcvf.f32.f16 q13, d18
87 154: f422437f vld2.16 {d4-d7}, [r2 :256]
88 158: f3f66723 vcvf.f32.f16 q11, d19
89 15c: f3f6c704 vcvf.f32.f16 q14, d4
90 160: f3f64705 vcvf.f32.f16 q10, d5
91 164: f24ecdec vadd.f32 q14, q15, q14
92 168: f3f6c62c vcvf.f16.f32 d28, q14
93 16c: f2484de4 vadd.f32 q10, q12, q10
94 170: f3f68706 vcvf.f32.f16 q12, d6
95 174: f3f64624 vcvf.f16.f32 d20, q10
96 178: e1510004 cmp r1, r4
97 17c: e1a02000 mov r2, r0
98 180: f26411b4 vorr d17, d20, d20
99 184: f3f64707 vcvf.f32.f16 q10, d7
100 188: f26c01bc vorr d16, d28, d28
101 18c: f24a8de8 vadd.f32 q12, q13, q12
102 190: f3f68628 vcvf.f16.f32 d24, q12
103 194: f2464de4 vadd.f32 q10, q11, q10
104 198: f3f64624 vcvf.f16.f32 d20, q10
105 19c: f26821b8 vorr d18, d24, d24
106 1a0: f26431b4 vorr d19, d20, d20
107 1a4: f443037f vst2.16 {d16-d19}, [r3 :256]
108 1a8: e1a03001 mov r3, r1
109 1ac: 1affffe0 bne 134 <conv_operation_f16+0x134>
110 1b0: e28cc001 add ip, ip, #1
111 1b4: e28ee080 add lr, lr, #128 ; 0x80
112 1b8: e155000c cmp r5, ip
113 1bc: 1affffda bne 12c <conv_operation_f16+0x12c>
114 1c0: e2888001 add r8, r8, #1

```

```

115 1c4: e2877d16    add r7, r7, #1408    ; 0x580
116 1c8: e1590008    cmp r9, r8
117 1cc: e086600a    add r6, r6, sl
118 1d0: 1affffd2    bne 120 <conv_operation_f16+0x120>
119 1d4: e59d3004    ldr r3, [sp, #4]
120 1d8: e5934004    ldr r4, [r3, #4]
121 1dc: e59d3000    ldr r3, [sp]
122 1e0: e2833001    add r3, r3, #1
123 1e4: e1540003    cmp r4, r3
124 1e8: e58d3000    str r3, [sp]
125 1ec: 8affff8f    bhi 30 <conv_operation_f16+0x30>
126 1f0: e28dd014    add sp, sp, #20
127 1f4: e8bd8ff0    pop {r4, r5, r6, r7, r8, r9, sl, fp, pc}
128 1f8: e3510000    cmp r1, #0
129 1fc: 00833083    addeq r3, r3, r3, lsl #1
130 200: 10823083    addne r3, r2, r3, lsl #1
131 204: 0283300f    addeq r3, r3, #15
132 208: 12833015    addne r3, r3, #21
133 20c: 00832002    addeq r2, r3, r2
134 210: 10866303    addne r6, r6, r3, lsl #6
135 214: 00866302    addeq r6, r6, r2, lsl #6
136 218: eaafffb9    b 104 <conv_operation_f16+0x104>

```

Figure C.17: Sparse `conv_operation` method Assembly code for the F16 NEON implementation.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl